Advances in Intelligent Systems and Computing 561

Stuart H. Rubin
Thouraya Bouabana-Tebibel *Editors*

Quality Software Through Reuse and Integration

Springer

Advances in Intelligent Systems and Computing

Volume 561

Series editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland
e-mail: kacprzyk@ibspan.waw.pl

Stuart H. Rubin · Thouraya Bouabana-Tebibel
Editors

Quality Software Through Reuse and Integration

Springer

Editors
Stuart H. Rubin
SPAWAR Systems Center Pacific
San Diego, CA
USA

Thouraya Bouabana-Tebibel
Laboratoire de Communication dans les Systèmes Informatiques
Ecole nationale Supérieure d'Informatique
Algiers
Algeria

ISSN 2194-5357 ISSN 2194-5365 (electronic)
Advances in Intelligent Systems and Computing
ISBN 978-3-319-56156-1 ISBN 978-3-319-56157-8 (eBook)
DOI 10.1007/978-3-319-56157-8

Library of Congress Control Number: 2017947506

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The idea of improving software quality through reuse is not new. After all, if software works and is needed, just reuse it. What is new and evolving is the idea of relative validation through testing and reuse and the abstraction of code into frameworks for instantiation and reuse. Literal code can be abstracted. These abstractions can be made to yield similar codes, which serve to verify their patterns. There is a taxonomy of representations from the lowest level literal codes to their highest level natural language descriptions. The quality of software is improved in proportion to the degree of all levels of reuse.

Any software, which in theory is complex enough to allow for self-reference, cannot be assured to be absolutely valid. The best that can be attained is a relative validity, which is based on testing. Axiomatic, denotational, and other program semantics are more difficult to verify than the codes, which they represent! But, is there any limit to testing, and how does one maximize the reliability of software through testing? These are the questions, which need to be asked. Here are the much sought-after answers.

Randomization theory implies that software should be broken into small coherent modules, which are logically integrated into the whole. These modules are designed to facilitate specification and may be optimized through the use of equivalence transformations. Moreover, symmetric testing (e.g., testing a sort routine using (3, 2, 1) (4, 3, 2, 1) (5, 4, 3, 2, 1)…) has minimal value beyond the first test case. Rather, the test cases need to cover all combinations and execution paths, which defines random-basis testing (e.g., (3, 2, 1) (3, 1, 2) (1) (2, 1, 2)…).

It is virtually impossible to generate a complete random-basis test for complex software. Hence, reuse in diverse applications serves as a substitute. Also, the more coherent the software modules, the greater their potential for reuse and integration, and the proportionately less the propensity for software bugs to survive as a consequence. Such an approach implies not only the use of integration testing, but the definition and application of true optimizing compilers as well.

For example, routines for reading data, iteratively bubbling the data to the list head to create a lexicographic order, and printing the result of the iterations (e.g., the *O* (n**2) bubble sort) can be iteratively transformed, by an optimizing compiler, into an O (n log n) sort (e.g., Quicksort). Such coherent optimizations may be self-referentially applied to the optimizing compilers themselves for further efficiencies of scale.

The aforementioned ways for optimizing literal software quality extend to all of its abstract patterns—only here testing applies to framework instances. It can be seen that software quality is properly dependent upon the scale of the effort. Scale, in turn, is defined by the embodied knowledge (i.e., including frameworks) and processor power, which can be brought to bear. Randomization theory and applicative experience converge as software is written for coherency, integrated for functionality, optimized for efficiency, executed to realize its latent potential for massive parallelism, and reused in diverse applications to maximize its validity (i.e., through random-basis testing).

One final note is appropriate. It should be possible to create domain-general higher-level (i.e., context-sensitive) languages through the inclusion of dialog, for disambiguation, and the capture of knowledge in the form of software frameworks. The higher the level of software, the more reusable it is on account of one central human factor—readability. That is why components are reusable—because their descriptive characterizations are understandable. This also follows from randomization theory although this theory does not address how to achieve such context-sensitive languages in practice—only the result of doing so.

Combining the key points herein leads to the inescapable conclusion that software productivity—including quality attributes—is not bounded, combines the best of theory and practice, and when realized, as described, will transform the software industry, as we know it, for the better.

The traveling salesman problem (TSP) is one of the most studied problems in optimization. If the optimal solution to the TSP can be found in polynomial time, it would then follow that every *NP-hard* problem could be solved in polynomial time, proving P=NP. In Chapter 1, it will be shown that the proposed algorithm finds $P \sim NP$ with scale. Machine learning through self-randomization is demonstrated in the solution of the TSP. It is argued that self-randomizing knowledge bases will lead to the creation of a synthetic intelligence, which enables cyber-secure software automation. The work presented in Chapter 2 pertains to knowledge generalization based on randomization. Inductive knowledge is inferred through transmutation rules. A domain-specific approach is properly formalized to deal with the transmutation rules. Randomized knowledge is validated based on the domain user expertise.

In order to improve software component reuse, Chapter 3 provides a mechanism, based on context-sensitive code snippets, for retrieving components and showing how the retrieved components can be instantiated and reused. This approach utilizes semantic modeling and ontology formalisms in order to conceptualize and reverse engineer the hidden knowledge in library codes. Ontologies are, indeed, a standard in representing and sharing knowledge. Chapter 4 presents a comprehensive methodology for ontology integration and reuse based on various matching techniques. The approach is supported by an ad hoc software framework, whose scope is easing the creation of new ontologies by promoting the reuse of existing ones and automating, as much as possible, the entire ontology construction procedure.

Given ever-increasing data storage requirements, many distinct classifiers have been proposed for different data types. However, the efficient integration of these classifiers remains a challenging research topic. In Chapter 5, a novel scalable framework is proposed for a classifier ensemble using a set of generated adjudications based on training and validation results. These adjudicators are ranked and assembled as a hierarchically structured decision model. Chapter 6 presents a graph-based solution for the integration of different data stores using a homogeneous representation. Schemas are transformed and merged over property graphs, providing a modular framework.

In chapter 7, heterogeneous terminologies are integrated into a category-theoretic model of faceted browsing. Faceted browsing systems are commonly found in online search engines and digital libraries. It is shown that existing terminologies and vocabularies can be reused as facets in a cohesive, interactive system. Controlled vocabularies or terminologies are often externally curated and are available as a reusable resource across systems. The compositional reuse of software libraries is important for productivity. However, the duplication and modification of software specifications, for their adaptation, leads to poor maintainability and technical debt. Chapter 8 proposes a system that solves these problems and enables the compositional reuse of software specifications—independent of the choice of specification languages and tools.

As the complexity of the world and human interaction grows, contracts are necessarily becoming more complex. A promising approach for addressing this ever-increasing complexity consists in the use of a language having a precise semantics—thus providing a formal basis for integrating precise methods into contracts. Chapter 9 outlines a formal language for defining general contracts, which may depend on temporally based conditions. Chapter 10 outlines the design of a system modeling language in order to provide a framework for prototyping complex distributed system protocols. It further highlights how its primitives are well-matched with concerns, which naturally arise during distributed system design. An operational semantics for the designed language, as well as results on a variety of state-space exploration techniques, is presented.

Chapter 11 addresses the integration of formalisms within the behavior-driven development tooling, which is built upon semi-formal mediums for specifying the behavior of a system as it would be externally observed. It presents a new strategy for test case generation. Chapter 12 focuses upon model checking Z specifications. A Z specification is preprocessed, where a generic constant is redefined as an equivalent axiom, and a schema calculus is expanded to a new schema definition. Chapter 13 presents a parameterized logic to express nominal and erroneous behaviors—including faults modeling, given algebra and a set of operational modes.

The created logic is intended to provide analysts with assistance in considering all possible situations, which may arise in complex expressions having order-related operators. It enables analysts to avoid missing subtle (but relevant) combinations.

January 2017

Stuart H. Rubin
Thouraya Bouabana-Tebibel

Contents

On the Tractable Acquisition of Heuristics for Software Synthesis Demonstrating that P~NP

Stuart H. Rubin[1(✉)], Thouraya Bouabana-Tebibel[2], and William K. Grefe[1]

[1] Space and Naval Warfare Systems Center Pacific, San Diego, CA 92152-5001, USA
{stuart.rubin, william.grefe}@navy.mil

[2] Ecole nationale Supérieure d'Informatique, LCSI Laboratory, Algiers, Algeria
t_tebibel@esi.dz

Abstract. The Traveling Salesman Problem (TSP) was first formulated in 1930 and is one of the most studied problems in optimization. If the optimal solution to the TSP can be found in polynomial time, it would then follow that every *NP-hard* problem could be solved in polynomial time, proving P = NP. It will be shown that our algorithm finds $P \sim NP$ with scale. Using a $\delta - \varepsilon$ proof, it is straightforward to show that as the number of cities goes to infinity, P goes to NP (i.e., $\delta > 0$). This was demonstrated using a quadratic number of parallel processors because that speedup, by definition, is polynomial. A fastest parallel algorithm is defined. Six distinct 3-D charts of empirical results are supplied. For example, using an arbitrary run of 5,000 cities, we obtained a tour within 0.00001063 percent of the optimal using 4,166,667 virtual processors (Intel Xenon E5-1603 @ 2.8 GHz). To save the calculated extra 209 miles would take a quantum computer, the fastest possible computer, over (5,000!/(2**4,978 * 22!)) * 267,782 centuries. Clearly, the automated acquisition of heuristics and the associated $P \sim NP$ solutions are an important problem warranting attention. Machine learning through self-randomization is demonstrated in the solution of the TSP. It is also shown, in the small using property lists, for an inductive logic of abduction. Finally, it is argued that self-randomizing knowledge bases will lead to the creation of a synthetic intelligence, which enables cyber-secure software automation.

Keywords: Heuristics · NP-Hard · $P \sim NP$ · Randomization · Software automation · Synthetic intelligence · TSP

1 Introduction

SSC-Pacific is constructing and testing a new approach to cybersecurity and automatic programming for its cost-effective realization. The approach has already proven to be effective for a game-theoretic program [1]. Here, the intent is to show that it can work in the absence of deterministic feedback. Thus, equivalent semantics, selected at

© Springer International Publishing AG 2018
S.H. Rubin and T. Bouabana-Tebibel (eds.), *Quality Software Through Reuse and Integration*, Advances in Intelligent Systems and Computing 561,
DOI 10.1007/978-3-319-56157-8_1

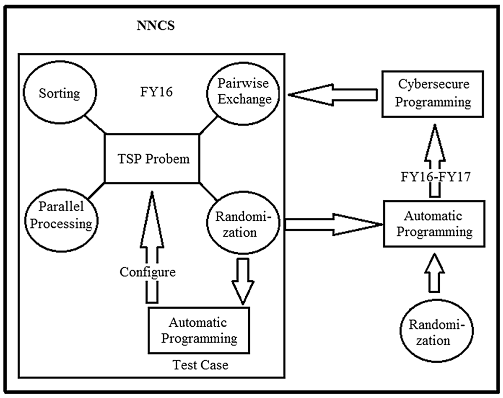


Fig. 1. Role of the TSP in cybersecurity and automatic programming

random, will be demonstrated to be cyber-secure. This makes it possible to protect all but purely and inherently random codes (e.g., the Mersenne Twister).

1. The core Traveling Salesman Problem (TSP) was selected because not only is it intractable – requiring a heuristic solution, but for this reason is notoriously difficult to cyber-secure (Fig. 1).
2. Our solution to the TSP problem involves highly-efficient sorting and comparison with a random path for Phase I. Alternatively, a comparison of two random walks may be elected (e.g., where a nearest-first solution is known not to be or not expected to be optimal) for Phase I. In Phase II, the better solution, found in the previous phase, is iteratively improved by way of pairwise exchange. These components are amenable to parallel processing and have already demonstrated a 94 percent improvement over the intuitive nearest-first solution using random data! They will be manually cyber-secured; and, will allow a user to attempt to infect them and/or steal cycles. It should not be possible to do this; and, early detection of attack is another expected feature.
3. A methodology, for creating and programming semantic code variants, will need to be automated to minimize the cost of cyber protection. This will be possible using randomization-based cyber-secure symmetric automatic functional programming.
4. Cyber security and automatic programming, in general, are predicated on the science of randomization. This essentially means reusing and transforming that which worked in the past and adapting it for the current situation. Novel requirements are satisfied by random codes. Everything else is related by transformation, which entails heuristic search. Heuristic search is realized by effective computer codes. The P $\sim$ NP problem is fundamental to tractable automatic programming. Thus, the larger the heuristic system, the more cost-effective it becomes. It follows that the near-optimal heuristic solution to the TSP problem can have enormous impact, in this regard, down the line.

Our belief is that cyber-attacks are spurring-on the development of automatic programming for cyber security as well as for automation in general. The impact here goes far beyond cyber security; and, it includes autonomous vehicles, fighting machines, design machines, and everything in between. It is good science and only good science that can bring these desirable outcomes about. It all starts with randomization-based cyber security.

2 The Heuristic TSP Algorithm

The best-found recursively enumerable heuristic solution to the TSP problem follows. It empirically demonstrates that the set of problems solvable in polynomial time is as close as desired, other than strictly equal to, the set of problems not solvable in polynomial time, or $P \sim NP$.

1. The TSP asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? Even though the problem is NP-complete, heuristic methods are known, which enable problems even with millions of cities to be approximated within a small fraction of 1 percent (e.g., within 0.05% of the optimal solution for the world tour problem). Again, for 5,000 cities (the limits of our PC's memory) we obtained a tour within 0.00001063 percent of the optimal. It is reported that for many other instances with millions of cities, solutions can be found that are within 2–3 percent of an optimal tour. Removing the condition of visiting each city "only once" does not remove the NP-hardness, since it is easily seen that in the planar case there is an optimal tour that visits each city only once (otherwise, by the triangle inequality, a shortcut that skips a repeated visit would not increase the tour length).
2. Any problem in the NP class can actually be reduced to the decision version of the travelling salesman problem. This means that any algorithm that solves the decision travelling salesman problem can be translated into one that solves all other problems in the NP class. Thus, suppose that one finds a polynomial-time algorithm for the decision travelling salesman problem. This would then mean that every problem in NP could be solved in polynomial time. This would prove that P = NP. Then, one could collect the offered $1 million prize from the Clay Mathematics Institute (i.e., $\delta = 0$). Again, we do not think this possible. Rather, it will be shown that our algorithm finds $P \sim NP$, $\delta > 0$ with scale, which is charted in the results section below.
3. Initialize the number of cities, n, and populate the distances between each pair of cities at random – between [1, n*(n-1)/2] miles (corresponding to the number of undirected edges). The distance between an arbitrary pair of cities need not be unique. Some non-symmetric duplication among the inter-city distances anneals the TSP problem by making it more complex to heuristically solve. This is the minimum number (i.e., corner point) allowing a distinct mileage assignment to each undirected edge.
4. The Quicksort algorithm is used to order the cities. It is faster than merge sort because the distances are stored in RAM. A test of 1,000,000 random elements was sorted in 156 ms by Quicksort vs. 247 ms for Merge sort. Quicksort performs well if the data is random. We maximize its speed by using the iterative (not recursive)

version of Quicksort, where arrays of size less than 17 (based on empirical results) are sorted using insertion sort. The sort is used to order the inter-city distances from nearest to furthest in a second array (see sample runs in the results section below). A starting city is selected at chance, which will be the same as the goal city. The nearest-first tour begins at the starting city and follows the path of minimal distance in the selection of each successive un-visited city. Ties are resolved by iterative optimization – in the second or heuristic phase.

5. Cities are tested for having been visited in sequence from nearest to furthest. The nearest not visited and previously visited cities are found by comparing what is stored at a vector position. If vector [position] = position, then city = position was visited – otherwise not. The first column is skipped as it refers to the minimal distance, which is from a city to itself. Equidistant cities are resolved FIFO (this will be optimized in the second phase of search). Visited cities are stored, in sequence, in a second vector. Upon conclusion, the residue, which comprises this vector, defines a random tour for purposes of comparison. That is, the nearest-first and random tours are compared and the shorter one serves as the initial iterate for Phase II. When all of the cities have been visited, return to the starting city and exit – completing the first phase of the tour. Again, the best-found tour is initialized by the minimum of the random tour and the nearest-first tour. The second-best tour is set to the remaining tour. These assignments appear to be equally likely in the absence of characterizing knowledge. An option allows for the use of a second random tour in lieu of the nearest-first tour. This option is faster and is to be selected whenever a nearest-first tour is known to be suboptimal (e.g., when the cities are not arranged like the equidistant end-spokes on a bicycle wheel). Selecting a good initial iterate in Phase I will speed-up the solution of the TSP in Phase II.
6. The required time to find an optimal solution can be estimated from n! tours based on the known time that it takes to do m tours, where m << n!. This allows much larger TSP problems to be estimated for optimality than could otherwise be searched. It was found that an optimal tour of 8 cities takes about 16 s, which is about the same time as a strictly random heuristic tour of 700 cities. Such search is based on n = 8 nested for loops running on one Intel Xenon E5-1603 chip @ 2.8 GHz. Random search provides a better estimate of the optimal solution, where the solution space is much larger than can be explored because it uniformly samples the space. The optimal tour is based on the cyber-secure differential (i.e., the optimal tour may be expected to be the minimal percentage improvement between the pairs of iteratively-improved tours better than the best-found tour). If the distinct second-best tour covers the same distance as the best-found tour, then the tours are said to be optimal. These tours are discovered using heuristic (subinterval) optimization as they would otherwise be intractable. (Note that the identity, $\log_2 x = (\log_{10} x)/(\log_{10} 2)$, where the bases and x are positive real numbers and the bases are not equal to one, may be used for calculations).
7. The system is allowed to iterate, without yielding an improved tour, for a defined time or number of iterations. It was found that about five million iterations is appropriate for 5,000 cities, or equivalently 1,000 * |cities|. This reduced the tour distance by 94 percent over Phase I. The limiting defined time or iterations is configured using NL for exploration. A starting and ending city are randomly

selected from [1, |cities-1|], where the starting and goal city are immutable. The entrance and exit distances, for the pair of cities, are subtracted from the best tour distance. The random pair of cities is swapped (i.e., pairwise exchange) and the entrance and exit distances, for the swapped pair of cities, are added to the best tour distance to replace the subtracted distances. If the resulting tour covers less distance, a new next-best tour and best tour are saved. Otherwise, the swap is reversed to restore the evaluation vector for the best found tour so far; and, the process of swapping cities begins anew. No relaxation technique can be faster.

8. The capability for iterative parallel heuristic optimization is what sets this algorithm apart from other good solution methodologies. Random pairwise exchanges and comparisons require semaphore protection for two groups of three-contiguous cities, or six cities at a time. However, the overhead associated with assigning, tracking, and releasing these blocks will approximately double the swap time. Given, n cities, this SIMD speedup is thus given as $\lfloor n / 2 \times 6 \rfloor$. Thus, for 120 cities for example, up to 20 SIMD processors will allow for a speedup by a factor of ten. A MIMD speedup may be included on top of the SIMD speedup. Here, up to n! processors (realistically n processors) independently solve the problem, where the overhead occurs in communicating the best solution found with the other processors. This communication overhead is O(n). Thus, the MIMD speedup will be a factor of n-1 times. Statistical redundancy is given by $n! / 2(n-2)!$. Given 5,000 cities, only one in 12,497,500 pairwise exchanges will be redundant as a result, which is insignificant. Combining SIMD and MIMD speedups, using $\lceil n^2 / 6 \rceil$ processors, the maximum realistic parallel speedup is $\lfloor n(n-1) / 12 \rfloor$, which is $O(n^2)$. The use of more processors can further reduce the runtime, but with ever-decreasing efficiency.
9. Cyber-Secure Application Note: The small subinterval problems are run using the same diverse semantically equivalent algorithms as for the larger intervals. However, the selected random order of cities is passed to semantically diverse algorithms to check cyber security as the sub-problems arise (i.e., different codes must produce identical outputs as is the case with Quicksort and insertion sort for example). Note that the symmetric half of these algorithms can be automatically programmed. They may steal up to half of the computational resources. The goal is to tractably check cyber security at defined frequent intervals. The minimum and maximum subinterval sizes are parameterized and configured using natural language – demonstrating our automatic symmetric programming methodology on a small scale. The smaller the interval, in general, the more likely optimizations are to be found (i.e., down to some minimal block size) and the more reliable the cybersecurity (i.e., since checks for cyber-attacks will be proportionately more frequent).

3 TSP Results

3.1 Graphic Presentation

The six charts below summarize, in visual form, the results of running the above algorithm on different hardware substrates. They make it clear that heuristic optimization and parallel hardware are fundamental to scientific and engineering advancement – something, which extends far beyond the confines of optimization and even cybersecurity and is based on symmetric automatic heuristic programming.

To begin, the time taken for the heuristic and optimal tours running on the Intel Xenon E5-1603 chip @ 2.8 GHz, a parallel machine based on the number of cities running the heuristic and optimal tours, and a quantum computer based on the number of cities (i.e., a 2^n theoretical speedup, which will be practically reduced) running the optimal tour – showing that $P \sim NP$, are compared. The computed distances from optimality are also shown. (Note that quantum computers might be able to factor large numbers, which is not possible using heuristic methods. Furthermore, $n! / 2^n = (n/2)! \times (n - 1/2)!$, where $3.5! = 3.5 \times 2.5 \times 1.5 \times 0.5$).

Figure 2 presents the runtimes for the above algorithm running on an Intel Xenon E5-1603 chip @ 2.8 GHz. These runtimes are on the order of two seconds for up to 100 cities. It then jumps to 42 s for 500 cities, 291 s for 1,000 cities, and 13,710 s for 5,000 cities. This characteristic quadratic climb in runtimes can be completely offset by having on the order of a quadratic number of processors, which is not always practical, or by relaxing the optimality requirements from a small fraction of one percent to say a few percent, or any combination thereof. Still, the spreadsheet shows that for just 17 cities, it will take 3.705075E + 10 times as long to compute the optimal as the heuristic tour on the same machine! Clearly, heuristic methods are inherently necessary.

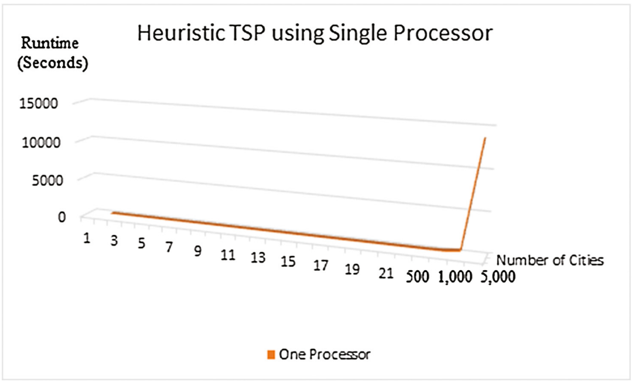


Fig. 2. Single processor heuristic runtimes for the TSP

Figure 3 presents the runtimes required by the above algorithm running on a (massively parallel) collection of virtual Intel Xenon E5-1603 chips @ 2.8 GHz. These runtimes are on the order of half a second using 6 processors for 6 cities, half a second using 9 processors for 7 cities, 1/8th of a second using 17 processors for 10 cities, 1/12th of a second using 38 processors for 15 cities, 1/50th of a second using 81 processors for 22 cities, 1/500th of a second using 1,667 processors for 100 cities, 1/2,000th of a second using 41,667 processors for 500 cities, 1/600th of a second using 166,667 processors for 1,000 cities, and 1/150th of a second using 4,166,667 processors for 5,000 cities. The spreadsheet shows that the runtime non-uniformly goes towards zero as the number of processors grows quadratically as defined by the number of cities.

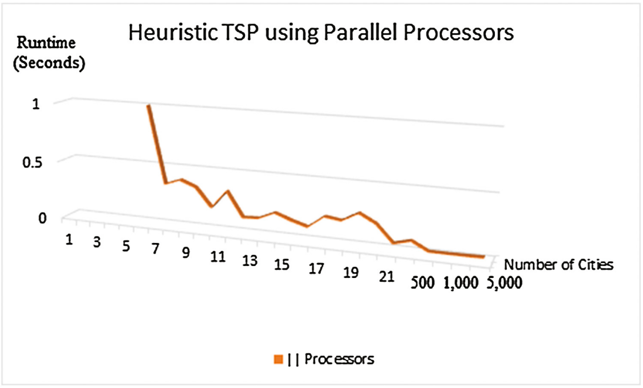


Fig. 3. Parallel processor heuristic runtimes for the TSP

Figure 4 shows that as the number of cities increases, the heuristic algorithm converges on the increasingly-near optimal solution. Even here, this is not practical for single processors as the number of cities grows without bound and is only practical using parallel processors, which can grow quadratically as a function of the number of cities. Again, the program allows optimality to be sacrificed for the sake of tractability should a sufficient number of processors not be available. The deviation from optimal is 7.7 percent for 5 cities, 10.42 percent for 7 cities, 3.4 percent for 10 cities, 0.9 percent for 15 cities, 0.09 percent for 22 cities, 0.01 percent for 100 cities, 0.0009 percent for 500 cities, 0.00007 percent for 1,000 cities, and 0.00001 percent for 5,000 cities.

Figure 5 presents the runtimes required by the O (n!) optimal algorithm running on an Intel Xenon E5-1603 chip @ 2.8 GHz. The runtime is 16 s for 8 cities, 151 s for 9 cities, 1,512 s for 10 cities, 16,632 s for 11 cities, 199,584 s for 12 cities, 2,594,592 s for 13 cities, 36,324,288 s for 14 cities, 544,864,320 s for 15 cities, 5,564,229,120 s for 16 cities, 1.48203E + 11 s for 17 cities, and (5,000!/20!) * 321,445 centuries, 6 years, and 334 days to compute for 5,000 cities. This characteristic exponential climb

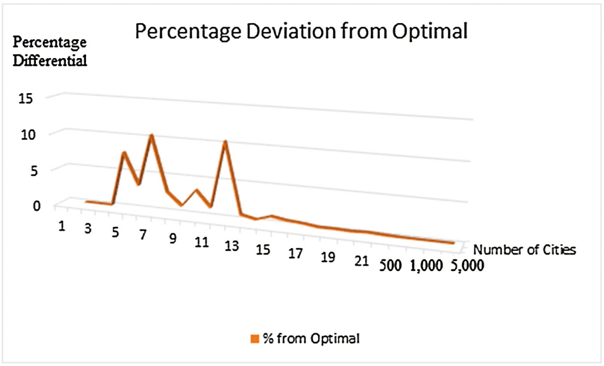


Fig. 4. Optimality of the heuristic runs

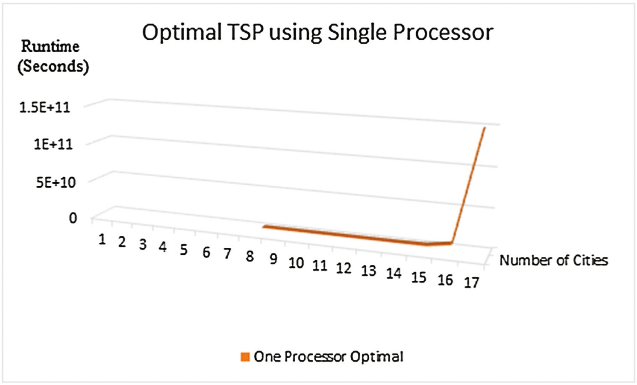


Fig. 5. Single processor optimal runtimes for the TSP

in runtimes can only be offset by relaxing the optimality requirements from a small fraction of one percent to say a few percent. Again, heuristic methods are inherently necessary.

Figure 6 presents runtimes required by the O (n!) optimal algorithm running on a (massively parallel) collection of virtual Intel Xenon E5-1603 chips @ 2.8 GHz. The runtime is 4 s using 11 processors for 8 cities, 25 s using 14 processors for 9 cities, 216 s using 17 processors for 10 cities, 1,848 s using 21 processors for 11 cities, 18,144 s using 24 processors for 12 cities, 199,584 s using 29 processors for 13 cities, 2,421,619 s using 33 processors for 14 cities, 32,050,842 s using 38 processors for 15 cities, 435,891,456 s using 43 processors for 16 cities, 6,736,504,320 s using 49 processors for 17 cities, and (5,000!/20!) * 15 years, 157 days, 20 h, 18 min, and 12 s to compute using 4,166,667 processors for 5,000 cities. The spreadsheet shows that while the runtime is less by a factor of 22 using parallel processors for 17 cities and by a factor of 2,082,933 using parallel processors for 5,000 cities, it nonetheless increases exponentially with the number of cities. This is expected.

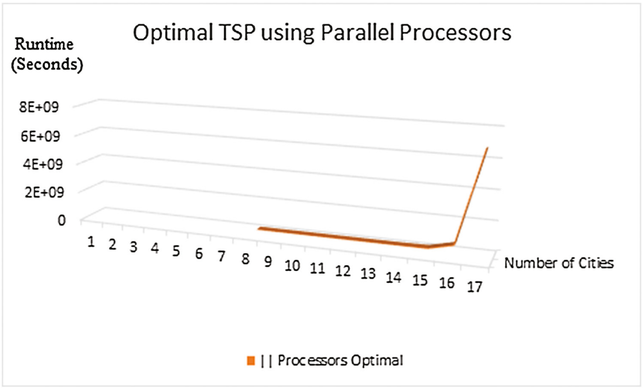


Fig. 6. Parallel processor optimal runtimes for the TSP

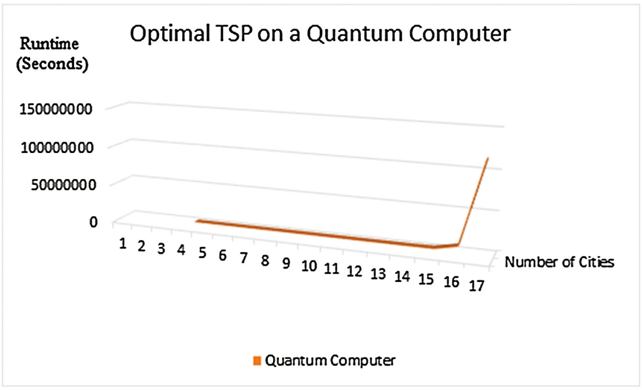


Fig. 7. Quantum computer optimal runtimes for the TSP

Figure 7 presents runtimes required by the O (n!) optimal algorithm running on a quantum computer, where the number of qbits equals the number of cities. Note that this is not meant to imply that such a machine will be built or even can be built. The runtime is less than 1 s for less than 10 cities, 5 s for 10 cities, 37 s for 11 cities, 309 s for 12 cities, 3,014 s for 13 cities, 34,286 s for 14 cities, 450,013 for 15 cities, 6,750,204 s for 16 cities, 114,753,474 s for 17 cities, and (5,000!/(2**4,978 * 22!)) * 267,782 centuries, 20 years, 269 days, 7 h, 6 min, and 41 s to compute for 5,000 cities. The spreadsheet shows that while the runtime is less by a factor of 59 than for parallel processors for 17 cities and by a factor of >> 1.0E + 99 using a 5,000-qbit quantum computer for 5,000 cities, it nonetheless increases exponentially with the number of cities. Again, this does not differ from expectations.

Next, we present actual executions to demonstrate the workings of the algorithm. All text is computer generated.

3.2 Five-City Tour

Fri Feb 12 08:10:25 2016

WELCOME TO THE CYBER SECURE AMERICAN AUTOMATED HEURISTIC TRAVELING SALESMAN

Enter the number of distinct cities (i.e., between 2 and 5,000)
? 5

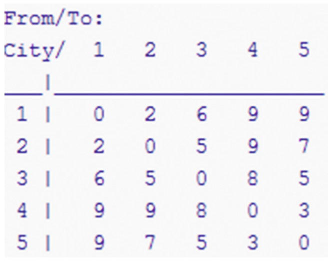
The tour will be comprised of 5 distinct cities numbered from 1 to 5.

Without loss of generality, the distance between the arbitrary pairs of 5 distinct cities is set to randomly vary between 1 and 10 miles (or equivalent).

The starting and ending city will be city number 1.

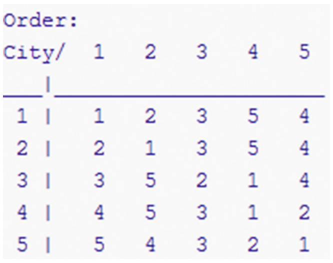
STARTING PHASE I OPTIMIZATION:

CITY TRAVERSAL MILEAGE (OR EQUIVALENT)



```
From/To:
City/  1   2   3   4   5
___|____________________
 1 |   0   2   6   9   9
 2 |   2   0   5   9   7
 3 |   6   5   0   8   5
 4 |   9   9   8   0   3
 5 |   9   7   5   3   0
```

FROM NEAREST (LEFTMOST) TO FURTHEST (RIGHTMOST) CITY



```
Order:
City/  1   2   3   4   5
___|____________________
 1 |   1   2   3   5   4
 2 |   2   1   3   5   4
 3 |   3   5   2   1   4
 4 |   4   5   3   1   2
 5 |   5   4   3   2   1
```

The distance covered by the Phase I nearest-first tour is 24 miles.
The distance covered by the Phase I random tour is 30 miles.
The Phase I nearest-first tour covers the minimal distance.
The nearest-first tour covers 80 percent of the distance covered by the random tour.

STARTING PHASE II ITERATIVE OPTIMIZATION:

Fri Feb 12 08:10:28 2016
The optimized tour of city numbers follows from left to right.

{START, 1, 2, 3, 5, 4, 1, GOAL}

The distance covered by the optimized tour is 24 miles.
Phase II optimization was not able to improve upon the results of Phase I.
The optimized tour is predicted to be an optimal tour.
This completes the tour of 5 cities.

On this computer, the heuristic optimization of the TSP for a 5-city tour took, 2 s to compute.

Using this algorithm, an optimal 5-city tour, on a 5-qbit quantum computer, will take 12.50000000 ms(s) to compute.

3.3 Twenty-Two-City Tour

Fri Feb 12 08:14:31 2016

WELCOME TO THE CYBER SECURE AMERICAN AUTOMATED HEURISTIC TRAVELING SALESMAN

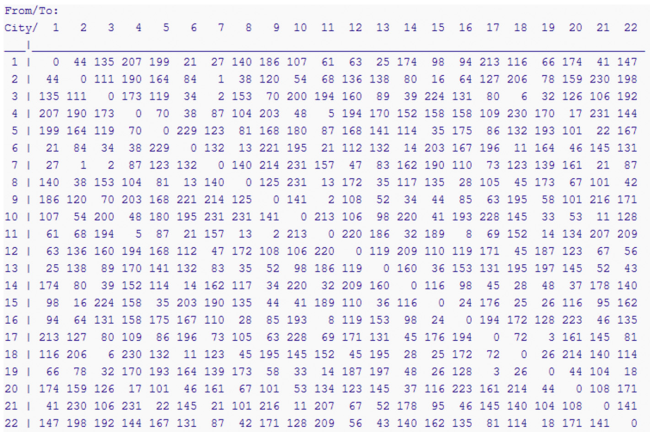
Enter the number of distinct cities (i.e., between 2 and 5,000)
? 22

The tour will be comprised of 22 distinct cities numbered from 1 to 22.
Without loss of generality, the distance between the arbitrary pairs of 22 distinct cities is set to randomly vary between 1 and 231 miles (or equivalent).
The starting and ending city will be city number 1.

STARTING PHASE I OPTIMIZATION:

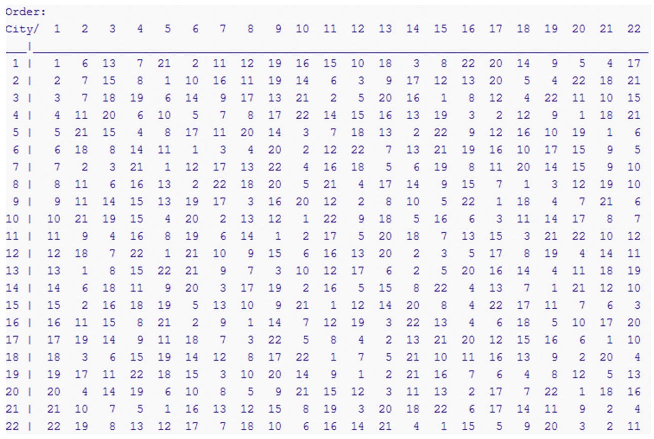
The distance covered by the Phase I nearest-first tour is 2,303 miles.
The distance covered by the Phase I random tour is 2,571 miles.
The Phase I nearest-first tour covers the minimal distance.
The nearest-first tour covers 90 percent of the distance covered by the random tour.

CITY TRAVERSAL MILEAGE (OR EQUIVALENT)



```
From/To:
City/  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22
___|________________________________________________________________________________________
 1 |   0  44 135 207 199  21  27 140 186 107  61  63  25 174  98  94 213 116  66 174  41 147
 2 |  44   0 111 190 164  84   1  38 120  54  68 136 138  80  16  64 127 206  78 159 230 198
 3 | 135 111   0 173 119  34   2 153  70 200 194 160  89  39 224 131  80   6  32 126 106 192
 4 | 207 190 173   0  70  38  87 104 203  48   5 194 170 152 158 158 109 230 170  17 231 144
 5 | 199 164 119  70   0 229 123  81 168 180  87 168 141 114  35 175  86 132 193 101  22 167
 6 |  21  84  34  38 229   0 132  13 221 195  21 112 132  14 203 167 196  11 164  46 145 131
 7 |  27   1   2  87 123 132   0 140 214 231 157  47  83 162 190 110  73 123 139 161  21  87
 8 | 140  38 153 104  81  13 140   0 125 231  13 172  35 117 135  28 105  45 173  67 101  42
 9 | 186 120  70 203 168 221 214 125   0 141   2 108  52  34  44  85  63 195  58 101 216 171
10 | 107  54 200  48 180 195 231 231 141   0 213 106  98 220  41 193 228 145  33  53  11 128
11 |  61  68 194   5  87  21 157  13   2 213   0 220 186  32 189   8  69 152  14 134 207 209
12 |  63 136 160 194 168 112  47 172 108 106 220   0 119 209 110 119 171  45 187 123  67  56
13 |  25 138  89 170 141 132  83  35  52  98 186 119   0 160  36 153 131 195 197 145  52  43
14 | 174  80  39 152 114  14 162 117  34 220  32 209 160   0 116  98  45  28  48  37 178 140
15 |  98  16 224 158  35 203 190 135  44  41 189 110  36 116   0  24 176  25  26 116  95 162
16 |  94  64 131 158 175 167 110  28  85 193   8 119 153  98  24   0 194 172 128 223  46 135
17 | 213 127  80 109  86 196  73 105  63 228  69 171 131  45 176 194   0  72   3 161 145  81
18 | 116 206   6 230 132  11 123  45 195 145 152  45 195  28  25 172  72   0  26 214 140 114
19 |  66  78  32 170 193 164 139 173  58  33  14 187 197  48  26 128   3  26   0  44 104  18
20 | 174 159 126  17 101  46 161  67 101  53 134 123 145  37 116 223 161 214  44   0 108 171
21 |  41 230 106 231  22 145  21 101 216  11 207  67  52 178  95  46 145 140 104 108   0 141
22 | 147 198 192 144 167 131  87  42 171 128 209  56  43 140 162 135  81 114  18 171 141   0
```

FROM NEAREST (LEFTMOST) TO FURTHEST (RIGHTMOST) CITY



```
Order:
City/   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22
___|________________________________________________________________________________________
 1 |    1   6  13   7  21   2  11  12  19  16  15  10  18   3   8  22  20  14   9   5   4  17
 2 |    2   7  15   8   1  10  16  11  19  14   6   3   9  17  12  13  20   5   4  22  18  21
 3 |    3   7  18  19   6  14   9  17  13  21   2   5  20  16   1   8  12   4  22  11  10  15
 4 |    4  11  20   6  10   5   7   8  17  22  14  15  16  13  19   3   2  12   9   1  18  21
 5 |    5  21  15   4   8  17  11  20  14   3   7  18  13   2  22   9  12  16  10  19   1   6
 6 |    6  18   8  14  11   1   3   4  20   2  12  22   7  13  21  19  16  10  17  15   9   5
 7 |    7   2   3  21   1  12  17  13  22   4  16  18   5   6  19   8  11  20  14  15   9  10
 8 |    8  11   6  16  13   2  22  18  20   5  21   4  17  14   9  15   7   1   3  12  19  10
 9 |    9  11  14  15  13  19  17   3  16  20  12   2   8  10   5  22   1  18   4   7  21   6
10 |   10  21  19  15   4  20   2  13  12   1  22   9  18   5  16   6   3  11  14  17   8   7
11 |   11   9   4  16   8  19   6  14   1   2  17   5  20  18   7  13  15   3  21  22  10  12
12 |   12  18   7  22   1  21  10   9  15   6  16  13  20   2   3   5  17   8  19   4  14  11
13 |   13   1   8  15  22  21   9   7   3  10  12  17   6   2   5  20  16  14   4  11  18  19
14 |   14   6  18  11   9  20   3  17  19   2  16   5  15   8  22   4  13   7   1  21  12  10
15 |   15   2  16  18  19   5  13  10   9  21   1  12  14  20   8   4  22  17  11   7   6   3
16 |   16  11  15   8  21   2   9   1  14   7  12  19   3  22  13   4   6  18   5  10  17  20
17 |   17  19  14   9  11  18   7   3  22   5   8   4   2  13  21  20  12  15  16   6   1  10
18 |   18   3   6  15  19  14  12   8  17  22   1   7   5  21  10  11  16  13   9   2  20   4
19 |   19  17  11  22  18  15   3  10  20  14   9   1   2  21  16   7   6   4   8  12   5  13
20 |   20   4  14  19   6  10   8   5   9  21  15  12   3  11  13   2  17   7  22   1  18  16
21 |   21  10   7   5   1  16  13  12  15   8  19   3  20  18  22   6  17  14  11   9   2   4
22 |   22  19   8  13  12  17   7  18  10   6  16  14  21   4   1  15   5   9  20   3   2  11
```

STARTING PHASE II ITERATIVE OPTIMIZATION:

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 2,223 miles.
This iteration reduced the tour distance by 3.47372992 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 2,013 miles.
This iteration reduced the tour distance by 9.44669366 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,831 miles.
This iteration reduced the tour distance by 9.04123199 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,778 miles.
This iteration reduced the tour distance by 2.89459312 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,590 miles.
This iteration reduced the tour distance by 10.57367829 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,519 miles.
This iteration reduced the tour distance by 4.46540881 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,449 miles.
This iteration reduced the tour distance by 4.60829493 percent.

```
Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,434 miles.
This iteration reduced the tour distance by 1.03519669 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,396 miles.
This iteration reduced the tour distance by 2.64993026 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,211 miles.
This iteration reduced the tour distance by 13.25214900 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,160 miles.
This iteration reduced the tour distance by 4.21139554 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,056 miles.
This iteration reduced the tour distance by 8.96551724 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 1,030 miles.
This iteration reduced the tour distance by 2.46212121 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 906 miles.
This iteration reduced the tour distance by 12.03883495 percent.

Fri Feb 12 08:14:33 2016
The distance covered by the iteratively optimized tour is 857 miles.
This iteration reduced the tour distance by 5.40838852 percent.

Fri Feb 12 08:14:33 2016
The optimized tour of city numbers follows from left to right.

{START, 1, 12, 2, 7, 21, 5, 15, 9, 11, 16, 8, 13, 3, 18, 19, 22, 17, 14, 20, 10, 4, 6, 1,
GOAL}

The distance covered by the optimized tour is 857 miles.
The Phase I tour was optimized by 63 percent in Phase II.
The optimal tour is predicted to be 1.03519669 percent better, or 848 miles.
That represents a potential improvement of 9 miles.
This completes the tour of 22 cities.

On this computer, the heuristic optimization of the TSP for a 22-city tour took, 1 s
to compute.
Using 81 processors, the runtime for this heuristic tour of 22 cities will take
48.94736842 ms(s) to compute.
```

On this computer, an optimal 22-city tour, of the TSP, will take (22!/20!) * 321,445 centuries, 6 years, and 334 days to compute.

Using 81 processors, the runtime for an optimal tour of 22 cities will take (22!/20!) * 8,459 centuries, 8 years, and 28 days to compute.

Using this algorithm, an optimal 22-city tour, on a 22-qbit quantum computer, will take 267,782 centuries, 20 years, 269 days, 7 h, 6 min, and 41 s to compute.

4 A Synthetic Intelligence for Secure Software Construction

4.1 Code Automation for Information Dominance

It is becoming increasingly rare for software to run in isolated environments. Rather, the software of the future will need to support the fleet in distributed heterogeneous computing environments connected by secure networks. Insuring the quality of that software provides the fleet with Information Dominance. Conversely, we seek to deny adversaries the capability to depend on their software.

It follows that the automatic programming (software automation) of quality software is a military objective towards achieving Information Dominance. Randomization [1] can provide the means by which the human on the loop realizes a high-level specification in effective code. Conventional compilers are essentially context-free translators. They are thus realizable through the use of context-free grammars with the translation of a few context-sensitive features through the use of attribute grammars. Nevertheless, progress in automatic programming hinges on the use of context-sensitive specification languages (e.g., structured English). Such languages are effectively compiled through the actions of rule bases. In fact, the first compilers were rule-based.

The design of rule-based compilers is not a theoretical problem. Rather, it is an economical process for the acquisition of knowledge for these compilers that has evaded solution. It follows that the solution of the automatic programming problem, including attendant cybersecurity, can be reduced to the efficient and cost-effective solution of the knowledge acquisition bottleneck. However, the more complex the knowledge, the more its' efficient acquisition depends on the acquisition of a salient knowledge base. This (second) knowledge base serves as heuristics for guiding the acquisition of the first one.

4.2 Auto-Randomized Learning for the TSP

If knowledge did not beget knowledge, there would be no need for heuristics, which was shown early on by Dendral and Meta-Dendral to not be the case [6]. It logically follows that the focal problem here is how to apply knowledge-based systems to their knowledge-acquisition bottleneck. The solution of this problem necessarily entails self-reference with scale (e.g., daisy-chained self-reference). A practical implication here is that secure networks will play a critical role in facilitating daisy-chained self-reference.

Self-reference implies the use of heuristics to prevent inherent incompleteness [7, 8]. The science of randomization offers a new way forward in the application of knowledge bases to themselves in order to extend them. For a real-world example, consider the following set of five rules pertaining to the solution of the TSP (Fig. 8).

Rules are self-contained coherent algorithms containing one or more entrance pre-conditions, a coherent action, and one or more exit post-conditions, which are often used to update a blackboard. Next, R1, R3, and R4 may be applied to R2 to randomize the rule base and in so doing extend its capability for finding a best heuristic solution. Here is the self-referentially randomized rule base, consisting of just two rules (Fig. 9). It is more general (capable) than the base from which it was derived. Appeal is made to Church's Thesis [7] to show that the mechanics of self-referential randomization are effectively realizable. These mechanics can be extended by inductive and abductive discovery through the use of schema-based search and I/O constraints [9]. The implications for cyber-secure automatic programming follow from previous discussions.

R1: Visit the closest city on Open, resolving ties arbitrarily, and remove it to Closed. (Closed is used to insure that no city may be visited twice.) Increase the tour length by the distance to this city.

R2: Traverse to the closest city on Open and then to the closest city to it on Open. No city may be visited twice. Compute all path lengths. Visit cities on the shortest path length, resolving ties arbitrarily, and remove them to Closed. Increase the tour (length) by the shortest path (length).

R3 (heuristic pruning): If the search-ply, at any city, exceeds t, execute R1.

R4: If no city remains on Open, return to the starting city and exit with the found solution.

R5: Run rule base with t = 1, 2, 4, …, 2**n, fitting a curve using the Newton Forward Difference Formula until the predicted runtime for t exceeds the allowance.

Fig. 8. A simple rule base for a heuristic solution of the TSP

It follows from previous results that automatic programming and hence knowledge acquisition is an NP-hard problem. It also follows that the quadratic-time solution to this problem entails the use of heuristics. Furthermore, it has been empirically demonstrated that self-reference is the key to successful heuristic acquisition, based on transferring results for the n-puzzle [3]. However, the results shown above make it clear that while this works, it can be arbitrarily difficult to acquire a set of appropriate heuristics. It then follows that the search for such heuristics must be pruned through the application of appropriate knowledge, which is to state that self-reference is key.

The need for search heuristics is inherent. This follows because, like the TSP, automatic programming is NP-hard. Even a full-scale programmable quantum computer could do no better than reduce the search by O(2**n). The TSP, as previously discussed, is O(n!), where O(n!) >> O(2**n). This differential is even more pronounced for true

```
R1: While Open <> { }
      Begin
        While the search-ply ≤ t
          Begin
            Traverse to the closest non-visited city on Open.
            Resolve ties arbitrarily.
            Increase the path length by the distance to this city.
            Visited cities in a loop are treated the same as those on
          Closed.
          End
        Removed visited cities, on the minimal path length, to
      Closed. (Closed is used to insure that no city may be visited
      twice.)
        Resolve ties arbitrarily.
        Increase the tour (length) by this path (length).
      End
    Exit with the solution.

R2: Run rule base with t = 1, 2, 4, …, 2**n, fitting a curve using the
Newton Forward Difference Formula (interpolating polynomial) until
the predicted runtime for t exceeds the allowance.
```

Fig. 9. A more general self-randomized rule base

automatic programming, which is as complex as O(n**n), where O(n**n) >> O(n!). Heuristics, heuristic acquisition, and heuristic transference [3] are inherently better enablers of machine learning through self-randomization than can quantum computers be. However, this is not to be interpreted as an argument to preclude the development of quantum computers to the extent possible. Rather, it puts priorities in perspective.

Clearly, this self-randomization of a knowledge base cannot be limited to any domain-specific set of problems. It is evidently more general than that. If the domain is set to that of any type of knowledge-based software associate (i.e., software whose purpose is to help the programmer in creating software – including itself), then cyber-secure automatic programming will likewise benefit. This follows because if any NP-hard problem (e.g., the TSP) can be shown to benefit from the approach, it then follows that every NP-hard problem will likewise benefit (e.g., even the very NP-hard problem of automatic programming).

It follows from the results of [2] that knowledge can grow to be of unbounded density by process of generalization. Maximizing the quantity of applicable knowledge can be shown to minimize the mean time for discovery, while maximizing the mean quality of the knowledge, which may be discovered.

The attainable density of knowledge is also dependent upon its representation (e.g., the missionaries and cannibals problem) [4]. Translating the representation of knowledge, in general, requires search and control knowledge to constrain that search. It follows that search, self-reference, and randomization are inherent to automatic programming in the large. The result is that automation, and hence intelligence, derives from effective very-high level representations (e.g., natural language) of context-sensitive knowledge.

This is not knowledge acquisition as portrayed in the literature. Rather, it takes the view that candidate knowledge stems from abductive questions that ask, "What would have to be true to obtain a further state randomization?" For example, helium balloons rise and hydrogen balloons rise. What would have to be true for gasses less dense than air to cause a balloon to rise? The answer is that hot air balloons must rise. Conventional expert system shells do not have access to this knowledge, unless it is literally supplied. Here, one sees how self-referential randomization can be driven by question-asking and/or experiment – leading to the creation of new and valid knowledge, which can be quite different from anything in the knowledge base(s), but which may be abduced from it.

4.3 Auto-Randomization Through Property Lists

Figure 9 demonstrates that the process of self-randomizing a rule base can be quite arduous and thus require an economy of scale to be effective. Nevertheless, there are inductive logics of abduction, which can be quite effective for self-randomization – even in the small. One such inductive logic of abduction involves the normalization and randomization of property lists. We will use our balloon example to illustrate the basic workings of the abductive process as follows. Consider the following two rules and their associated property lists (Fig. 10).

Next, a fixed approximation of applying the rule base to itself, or self-randomization,

R1: If red hydrogen balloon, then it will rise.
R2: If orange helium balloon, then it will rise.
P1: Property (red) = {color, sunrise, sunset, blood, hot, …}
P2: Property (orange) = {color, fruit, hotter than red, …}
P3: Property (hydrogen) = {gas, flammable, less dense than helium, odorless, tasteless, …}
P4: Property (helium) = {gas, inert, less dense than air, odorless, tasteless, …}

Fig. 10. Rules and associated property lists

is to find for (a) R1(P1) ∩ R2 (P2) and (b) R1(P3) ∩ R2(P4). In a larger rule-base segment, there can be many more rules. Here, temporal locality, based on recentness or frequency of reference, is used to determine the candidate rules and properties to be intersected. For example, a child may study a blue helium balloon and then a red train with no randomization ensuing. Conversely, he/she might then study a red helium balloon and abduce that the color of the balloon is of no consequence by randomizing the associated rule base.

Intersection (a) shows that the property of having color is common to rising balloons. Intersection (b) shows that an odorless, tasteless, gas, which also has the (transitive) property of being less dense than air is common to rising balloons. The density property is arrived at through the use of a transitive logic (i.e., hydrogen < helium and helium < air → hydrogen < air). Thus, R1 and R2 are replaced (randomized) by:

R1’: If {color} and {odorless, tasteless, gas, less dense than air}, then it will rise.

At this point, suppose the child acquires the rule:

R3: If red air balloon, then it will fall.

and the associated property list:

P5: Property (air) = {gas, oxidizer, more dense than helium, odorless, tasteless, …}

In R3, red is found to have the property of color. However, R1’ and R3 have consequents, which are not subsets of one another. In fact, these consequents are contradictory. Thus, if the balloon rises or falls cannot be attributed to color alone. It must be due to some variance in the properties of air and the unified gas properties found in R1’. That is, the hydrogen/helium balloon may rise because it is less dense than air and the air balloon may fall because air is an oxidizer and/or more dense than helium. Formally, (c) = ((b) − R3(P5)) = {less dense than air} and (d) = (R3(P5) − (b)) = {oxidizer, more dense than helium}. At this point, we have the rules:

R1”: If {color} and {less dense than air}, then it will rise.
R3’: If {color} and {oxidizer, more dense than helium}, then it will fall.

You might remember, as a kid, seeing transparent helium balloons and being somewhat astonished that they would rise (R4). I know that I was. The property of transparent = {no color}. The self-randomization here is (e) = R1” (color) ∩ R4(no color) = Ri(). Thus, the property of color is dropped from all rules pertaining to rising balloons (i.e., R1” and R4). Similarly, experience with a transparent air balloon would result in the property of color being dropped from all rules pertaining to falling balloons (i.e., R3’).

Alternatively, general self-randomization allows for a (common-sense) rule that states if color does not cause a balloon to rise, then it does not cause a balloon to fall, to be applied and thus cancel the need for this last experiment. It should be noted that this rule is not at all obvious to a machine and would need to be acquired. After all, a lead weight does not cause a balloon to rise, but it will cause a balloon to fall. At this point, we have the rules:

R1’”: If {less dense than air}, then it will rise.
R3”: If {oxidizer, more dense than helium}, then it will fall.

Next, consider the process for asking questions, which is driven by the goal of obtaining a randomization. This too is a form of learning. Here, one might ask, “Are there any non-oxidizers more dense than helium; and, if so what would happen to a balloon filled with one?” The conveyed answer is that Freon (dichlorodifluoromethane) is such a substance; and, it causes the balloon to fall. This creates a tautology and thus the rule:

R3''': If {more dense than helium}, then it will fall.

Of course, this rule is not exactly correct as written. The following question, driven by randomization is thus posed, "Are there any gasses denser than helium that will not fall?" The conveyed answer is that pure nitrogen is denser than helium and will cause a balloon to rise in the air. This fact contradicts the rule; and, it thus leads to a question seeking to elicit the reason for a gas balloon falling; or, in the absence of such feedback, an iterative convergence on the same. Here, the next step in such a convergence would be:

R3'''': If {more dense than nitrogen}, then it will fall.

Alternatively, R1''' can serve as the basis of a more-informed question; namely, "If Not {less dense than air} will it fall?" A table lookup may or may not be available to show that Not {less dense than air} → {equal to or more dense than air}. This can also lead to the deduced property that pure nitrogen is less dense than air. Notice that as properties and rules are updated, there is a cascading of such similar updates.

At this point, we have the following rules.

R1''': If {less dense than air}, then it will rise.
R3''''': If {equal to or more dense than air}, then it will fall.

The novel question being posed for this rule base is what will happen if a balloon is filled with hot air? In order to answer this question, one needs to obtain the property list for hot air. If this is not available, or the specific property of density is not present, one needs to generate a query. This query is not directly for the purpose of randomization. Rather, it seeks to enable future randomizations, as well as higher-quality randomizations, through property extensions. Since hot air has the property of being less dense than air, the system can respond that hot air balloons will rise – despite having no literal rule to that effect.

Notice that as the system tends towards the capability for understanding natural language, it can interact with other knowledge-based segments in a daisy chain to answer, or at least learn to answer, questions that will be posed. These arguments provide a rather informal proof that a randomization-based synthetic intelligence is indeed possible. Note too how inquisitive the operational system becomes. This is not only very human-like, but suggestive that our own questions may indeed be driven by the need for randomization.

4.4 State and Representation Randomization

The case for representation is similar. For example, the representation of an image, in terms of edges, can greatly facilitate its correct characterization. Then, the representation of a human x-ray, in terms of edges, can similarly greatly facilitate a diagnosis. What would have to be true of a sonar image to facilitate its characterization? The answer is that its representation, in terms of edges, must facilitate its correct characterization. Here, the randomization of representation leads to the creation of new and valid knowledge. This knowledge is symmetric, which allows the abductive process of randomization to be applied. Again, such abduced knowledge can be very different from that embodied by the knowledge base(s).

Of course, state and representation randomization can co-occur. One will find that just because this is the theoretical answer, does not necessarily or adequately address its empirical realizations. The empirical KASER [5] was a first step in demonstrating that such an intelligent system – one enabling automatic programming, or intelligent problem solving in its own right, could be constructed with great practical success. It remains to develop technologies for representation randomization as well as more capable technologies towards unbounded state and representation randomization. Not only do these define true (i.e., synthetic) intelligence, but non-trivial software automation as well. Just as complex variables play a role in the solution of real-world equations, schemas coupled with I/O constraints can play a role in the randomization of real-world knowledge bases (e.g., for software associates).

4.5 On Self-randomization for Generalization

One of the problems with property sets is that combinations of variables are, in general, what is enabling generalization, which involves n choose r possibilities – exponential. Thus, while it is relatively simple (and eminently practical) to induce that lighter than air gasses are what causes a balloon to rise, it is far more arduous to determine say what are the properties of TiO2 that enable it to crack water in the presence of sunlight (i.e., in the search for better catalysts). Clearly, the only hedge against this inherent combinatorics problem is knowledge; and, the best way to acquire such knowledge is through self-randomization. Minton [10] applied knowledge to the inference of knowledge using explanation-based learning (EBL). However, EBL could not account for the context of where and when to properly apply that knowledge.

In the solution of this problem, we first provide a conceptual account of self-randomizing explanation-based learning (SREBL). Then, we will proceed to address some more formal aspects of the algorithm. SREBL conceptually operates as shown in Fig. 11.

The result of SREBL is not only maximally generalized domain-specific rules (e.g., rules for interpreting NL), but maximally generalized meta-rules for maximally generalizing domain-specific rules, including themselves, as well.

The utility of this approach follows from the combinatorics of rule application. If a generalization is defined to amplify rule applicability by a mean factor of m and the average segment comprises n rules, then the knowledge amplifies its utility by a factor of m^n. Furthermore, this is not a static factor, but m and n grow with rule acquisition. Thus, acquired knowledge grows exponentially!

At least two salient points need to be made. First and foremost, representation is defined and limited by the programming language used. It is possible to bootstrap that language (e.g., towards NL) to yield far more capable generalization rules. This provides for the attainment of unbounded state and representation randomization, or information-theoretic black holes [2]. Here, randomization is not due to information compression so much as it is attributable to the generalization of knowledge.

1. Start with a rule base of domain-specific rules. Fired rules are logically moved to the head from where they may be selected as candidates for generalization. Generalizations are context-sensitive tags based on evolving set memberships. The contexts provided by predicate instances are recursively defined by non-monotonic acquired rules.
2. Map this rule base to one comprised of more-general rules, ultimately written in a bootstrapped NL, which subsume their instantiations and are found by query. The presence of a single contradictory rule indicates an overgeneralization error. Subsumed rules may be logically moved to a dynamic cache, or simply expunged.
3. For each categorically new generalized rule, create a domain-specific meta-rule base segment, which is comprised of rules, ultimately written in a bootstrapped NL, which are found by query and iteratively generalize a domain-specific rule.
4. Meta-rule base segments are augmented with specific meta-rules to complete the generalization of new domain-specific rules. If no segment properly applies, a new segment is created and populated.
5. Take the meta-rule base segments to be the aforementioned domain-specific rules and map those from the logical head to their generalizations.
6. Self-referentially apply the meta-rule base segments to their own generalizations.
7. This process ends when no meta-rule base segment can be further generalized at this time.

Fig. 11. Boostrapping self-randomizing rule bases for generalization

Second, the previously demonstrated need to ask questions takes the form of iteratively asking the user (or another domain-specific segment) to supply a rule, in the programming language, which generalizes a rule, if possible, or which maps a domain-specific rule to its specified generalization. As the system learns, the sequence of queries becomes increasingly high-level and never redundant. This is attributable to randomization. The system must test that specified rules are not over-generalized, which can be autonomously detected because it results in contradiction. The system must also test that generalization rules are not overly general, which can be detected because they map a specific rule to one having a contradictory instance(s).

Biermann [11] at Duke University wrote a non-creative expert system that synthesized "impressive" LISP programs using rules. The main inefficiency was the knowledge-acquisition bottleneck, or rule-discovery process. Self-randomizing rule bases can mitigate that bottleneck and enable the creation of cyber-secure synthetic software. This software can be represented in the form of an expert system running coherent rules, or in the form of functional programs. The point is that not only are all manner of software associates possible, but the high-level codes themselves can also be synthesized with the human on the loop, which again follows from the work of Biermann [11].

Self-randomizing knowledge bases define more intelligent systems. These knowledge repositories will drive software associates and synthesizers, where the larger the scale, the more capable the system. Fortunately, a proof of concept is possible in the small.

If and when such machines become capable of unbounded state and representation randomization [2], they again will necessarily create information-theoretic black holes. We postulate that the physics of information may define self-awareness. Convergent creativity and intelligent life may very well be inseparable. Einstein was in agreement, based on his assertion, “Imagination is more valuable than knowledge.”

5 Conclusion

Running the algorithm for up to 5,000 cities, the Phase I tour was optimized by up to 94 percent in Phase II. This was found to be within 0.00001063 percent of optimal (i.e., 209 miles). This suggests, in decreasing order of likelihood, that (a) pairwise exchange cannot globally optimize the TSP solutions (e.g., Each of two adjacent pairwise exchanges would increase the global distance slightly, but taken together they would reduce it – however, finding such ordered pairs, ordered triplets, and so on is clearly intractable for exponentially diminishing returns. Hence, (c) follows.), (b) the random number generator has an abbreviated period and/or is not uniform, and/or (c) P = NP, where P is quadratic and the scale tends towards infinity. At least one thing is certain; namely, $P \sim NP$ for all practical applications, where P is quadratic. One need not await the awarding of the Clay $1 million mathematics prize to practically solve every NP-hard problem (again, not just the TSP) within quadratic time! Here, solving implies getting arbitrarily close to, but not necessarily at, optimal. While quadratic time is not practical for all applications, it opens the door for automatic program synthesis through heuristic search and discovery [3] – something of enormous practical import.

Machine learning through self-randomization was shown to be practical through inductive logics of abduction. Daisy-chained randomization benefits from scale and was shown in the discovery of heuristics for the practical solution of the NP-hard TSP. Furthermore, it was informally shown how an inductive logic of abduction could be developed using property lists to enable machine learning, in the small, through the self-randomization of the associated knowledge base(s). This capability is amplified by an allowance for query. Here, the capability for machine learning is proportionate to the capability for the knowledge base to randomize itself. As this capability grows, it ultimately defines an information-theoretic black hole [2]. This is what makes possible the development of a synthetic intelligence. A capability for unbounded cyber-secure software automation is reducible to this development.

Acknowledgements. Stuart would like to acknowledge the assistance of Professor Thouraya Tebibel and student Yasmin Hoadjli, for their work in independently verifying the results reported by this paper. Stuart would like to extend special thanks to Michael Leyton, Rutgers Department of Mathematics, for reviewing his “tremendously exciting” Theory of Randomization [2] and to the kind genius, Sukarno Mertoguno for applicative discussions on cybersecurity. Stuart would also like to extend his thanks to the Office of Naval Research (ONR) for providing financial backing for this research with the support of Ken Simonsen. This work was produced by a U.S. government employee as part of his official duties and is not subject to copyright. It is approved for public release with an unlimited distribution.

References

1. Rubin, S.H.: Randomization for Cyber Defense, Patent Disclosure, NC 103845, 28 October 2015
2. Rubin, S.H.: On randomization and discovery. Inf. Sci. **177**(1), 170–191 (2007)
3. Rubin, S.H., et al.: On heuristic randomization and reuse as an enabler of domain transference. In: Proceedings of the 2015 IEEE International Conference on Information Reuse and Integration (IRI), San Francisco, CA, 13–15 August 2015, pp. 411–418 (2015)
4. Amarel, S.: On representations of problems of reasoning about actions. Mach. Intell. **3**, 131–171 (1968)
5. Rubin, S.H., Murthy, S.N.J., Smith, M.H., Trajkovic, L.: KASER: knowledge amplification by structured expert randomization. IEEE Trans. Syst. Man Cybern. Part B Cybern. **34**(6), 2317–2329 (2004)
6. Mitchell, T.M., Carbonell, J.G., Michalski, R.S. (eds.): Machine Learning: A Guide to Current Research. Springer-Verlag Inc., New York (1986)
7. Kfoury, A.J., Moll, R.N., Arbib, M.A.: A Programming Approach to Computability. Springer, New York (1982)
8. Uspenskii, V.A.: Gödel's Incompleteness Theorem, Translated from Russian. Ves Mir Publishers, Moscow (1987)
9. Rubin, S.H., Tebibel, T.-B.: NNCS: randomization and informed search for novel naval cyber strategies. In: Abielmona, R., et al. (eds.) Recent Advances in Computational Intelligence in Defense and Security, Studies in Computational Intelligence, vol. 621, pp. 193–223. Springer 2016
10. Minton, S.: Learning Search Control Knowledge: An Explanation Based Approach. Kluwer International Series in Engineering and Computer Science, New York (1988)
11. Biermann, A.W.: Automatic programming: a tutorial on formal methodologies. J. Symbolic Comput. **1**(2), 119–142 (1985)

An Approach Transmutation-Based in Case-Based Reasoning

Thouraya Bouabana-Tebibel[1(✉)], Stuart H. Rubin[2], Yasmine Hoadjli[1], and Idriss Benaziez[1]

[1] Ecole nationale Supérieure d'Informatique, LCSI Laboratory, Alger, Algeria
{t_tebibel,y_hoadjli,i_benaziez}@esi.dz
[2] Space and Naval Warfare Systems Center Pacific, San Diego, CA 92152-5001, USA
stuart.rubin@navy.mil

Abstract. Case-Based Reasoning (CBR) interests the scientific community, whom are concerned with scalability in knowledge representation and processing. CBR systems scale far better than rule-based systems. Rule-based systems are limited by the need to know the rules of engagement, which is practically unobtainable. The work presented in this paper pertains to knowledge generalization based on randomization. Inductive knowledge is inferred through transmutation rules. A domain specific approach is properly formalized to deal with the transmutation rules. Randomized knowledge is validated based on the domain user expertise. The approach is illustrated with an example and is seen to be properly implemented and tested.

Keywords: Case-Based Reasoning · Contextual search · Randomization · Transmutation

1 Introduction

The performance of a Case-Based Reasoning (CBR) system is highly correlated with the number of cases that it imbues. This is evidenced by the resolution process which seeks the most similar cases found in the case base to solve new (sub-)problems. Each case represents a situational experience already saved by the system. Solutions related to the most similar problems are retrieved for reuse. They are proposed to the user who may revise them according to the current situation. The more similar the experience is to the encountered case (situation), the more likely an adaptation is to be valid. The case base is enriched by the acquisition of revised new cases.

The size of the knowledge base, as source of the primitive knowledge and as a warehouse to the generated cases, is worthy of consideration. On one hand, the more and the richer base, the more accurate the system is with regard to resolution. On the other hand, too large knowledge base slows down the reasoning process. The main purpose served by this paper is to define how to integrate knowledge generation in a CBR system.

Implicit knowledge can be present in the case base through dependencies and similarities between cases. When dependencies are strong, they allow for knowledge derivation by transformation based on equivalence. When they are weak, knowledge derivation is obtained by randomization based on analogy. In this paper, we define an approach to infer knowledge by materializing the implicit knowledge through newly-generated cases. Knowledge inference will be inductive and based on randomization.

Randomization is a new trend in CBR processing with many pioneering works by Rubin et al. [1–7]. These works show that hybridization of CBR and fuzzy methods leads to compromise solutions with respect to accuracy and computational complexity. Randomization is used for case retrieval in CBR.

However, recent work in Case-Based Reasoning [3, 4] has evidenced that while cases provide excellent representations for the capture and replay of plans, the successful generalization of those plans requires far more contextual knowledge than economically sound planning will allow. The approach to randomization, proposed herein, enables the dynamic extraction of transformative cases, which permits the inference of novel cases through the substitution of problem/solution cases along with a check of case validity. It is based on the work proposed in [6] which proposes an approach to derive knowledge by randomization using analogy among the segments of information. The approach proposed in [8] first extends the previous work with new conditions on the randomization process.

In this paper, we propose to formalize the cases semantics, to better handle randomization and validate the randomized cases. We also show how such a randomization may be applied to a case-based reasoning system, and how the latter may benefit from randomization with respect to the problem and solution parts, which define a given case.

An appropriate application illustrates the approach. It consists of a travelling robot. The robot has as its mission to move between two points with the obligation of passing through a number of checkpoints. Such an application finds utility in many areas, like road transportation and office environments. For the latter area, some tasks of the robot could be: mail delivery, document exchange between offices, guest orientation, etc.

The remainder of the paper is structured as follows. Section 2 presents works related to randomization, especially for case-based reasoning systems. Section 3 presents the proposed transmutation approach. Section 4 develops the technical aspect of the approach integration into a CBR system. Section 5 applies the approach to a specific domain and Sect. 6 validates it. Finally, Sect. 7 illustrates the approach through an appropriate application.

2 Related Work

Case-based reasoning has been useful in a wide variety of applications. Health science is one of its major application areas [9]. In addition, financial planning, decision making in general [10, 11], and the design field [12–14] are representative of important application areas for case-based reasoning.

Indeed, many CBR systems have been developed in the past two decades. While increased interest in the application of CBR is observed, the design and development process of the CBR system itself is still the subject of many studies in the literature. The focus is on the theories, techniques, models, algorithms and the functional capabilities of the CBR approach. For example, in the retrieval phase, authors of [15] argue that how to best design a good matching and retrieval mechanism, for CBR systems, is still a controversial issue. They aim to enhance the predictive performance of CBR and suggest a simultaneous optimization of feature weights, instance selection, and the number of neighbors that can be combined using genetic algorithms (GA). The best method to index cases is still an open issue. Fuzzy indexing is one of the approaches, which that have been successfully applied by previous researchers in different applications, such as presented by the works in [16, 17].

Generalization and fuzzification of similar cases comprises the basis for search improvement and case-base size reduction. Randomization is used for case retrieval in CBR. In [18–20] fuzzy similarity measures are applied to range and retrieve similar historical case(s). In [18], a fuzzy comprehensive evaluation is applied for the similarity measure within the CBR approach proposed for reusing and retrieving design information for shoes. In [21], a k-NN clustering algorithm is proposed.

Fuzzy logic is advantageous in knowledge transfer as shown in [22], where uncertainty and vagueness characterize the target domain. An example of the use of fuzzification can be found in [23]. In this work, a framework for fuzzy transfer learning is proposed for predictive modeling in intelligent environments.

Hybridization of multiple techniques is adopted in many applications. In [24] a hybrid decision model is adopted for predicting the financial activity rate. The hybridization is performed using case-based reasoning augmented by genetic algorithms (GAs) and the fuzzy k nearest neighbor (fuzzy k-NN) methods.

Randomization is also used in the reuse phase of CBR to make a decision on whether or not the most similar case should be reused. In [25], a random function is applied on a special category of users that have made similar requests to compute their mean opinion.

Similarly, in [26], decision making on knowledge handling within a CBR system is based on fuzzy rules – thus providing consistent and systematic quality assurance with improvement in customer satisfaction levels and a reduction in the defect rate. In [27], a scoring model based on fuzzy reasoning over identified cases is developed. It is applied to the context of cross-document identification for document summarization.

3 Transmutation Approach

A methodology for the induction of knowledge based on randomization is proposed in [6]. The methodology introduces an approach to derive knowledge by transmutation using analogy among the base cases. We dedicate, in this paper, the approach to an exclusive use in CBR systems. This application is given in the two following subsections, which respectively deal with the problem and solution transmutation.

In the CBR base, a case is represented by a grammatical production in the form of: $Problem\,Situation \rightarrow Solution\,Action$; where the left-hand side of the production expresses the problem part, and the right-hand side is the solution part. The case base is defined to be domain- specific. The problem is composed of multiple descriptors {A, B, C…} and the solution is composed of multiple components (X, Y, Z…).

Transmutation on the grammatical production is based on analogies between cases and provides new productions, thus enriching the case base. Two types of transmutation may be considered. The first one consists in a problem substitution; whereas the second one is a substitution of the solution part. Both types of transmutation require three specific cases, which we will call transmutation trio or trio for short.

3.1 Problem Substitution

The first type of transmutation is based on a trio composed of: (1) two cases with the same solution part, but different problem parts; they will be called primary cases; (2) a third case on which the substitution can apply; thus a case whose problem part includes the problem part of one of the primary cases. It is called a substitution case. For example, we consider the following trio:

C1 : $(A, B, C) \rightarrow (X)$

C2 : $(A, E) \rightarrow (X)$

C3 : $(A, B, C, D) \rightarrow (X, V)$

We note that C1 and C2 have the same solution (X). Thus (A, B, C) and (A, E) can substitute for each other in the context of (X), since they have the same solution part. We notice that (A, B, C) is included in the problem part of C3 and (X) in its solution part. From these cases we can derive a new case:

C3′ : $(A, E, D) \rightarrow (X, V)$

3.2 Solution Substitution

The second type of transmutation is based on a trio composed of: (1) two cases, called primary cases, with the same problem part, but different solution parts; (2) a third case, called substitution case, on which the substitution can apply; thus a case whose solution part includes the solution part of one of the primary cases. We consider, for example, the following trio:

$$\text{C3}' : (A, E, D) \rightarrow (X, V)$$

$$\text{C4} : (A, E, D) \rightarrow (U)$$

$$\text{C5} : (A, E, D, F) \rightarrow (U, T)$$

We note that C3′ and C4 are non-deterministic; the same problem (A, E, D) is mapped to distinct solutions. Thus (X, V) and (U) can substitute for each other in the context of (A, E, D), since they have the same problem part. The newly generated case C5′ will be:

$$\text{C5}' : (A, E, D, F) \rightarrow (X, V, T)$$

Note that these transformative substitutions are often, but not always valid. This follows because there can be complex latent contextual interactions. For example, it may be that my stove can burn alcohol and simple contextual transformations tell us that it should also burn kerosene. However, here the latent context is that kerosene is more viscous than alcohol, which serves to interfere with the fuel-injection mechanism. Clearly, such transformations require the application of domain-specific knowledge to insure validity.

4 Knowledge Induction in a CBR System

The process of knowledge generation based on the proposed approach serves to amplify the case base with new knowledge, thus enhancing the capability of search in CBR systems for delivering more potentially similar cases. Such knowledge amplification [2] relies on the built transmutation trios. The more the transmutation trios are well-structured the faster will be their retrieval. For this purpose, we propose a case-based segmentation driven by the trios.

4.1 Knowledge Structuration

To speed up retrieval of the transmutation trios, we structure the case base around the trio. We split the knowledge base into segments, where a segment serves to gather cases forming transmutation trios. Trios that belong to the same segment share, necessarily, at least one common case. Each segment is represented by a delegate. The delegate concerns only the problem part. It is a generic description that best represents problems within the same segment. It also can be defined as a generalization yielding the most significant part of a problem. A case is added to the segment if its problem matches the delegate.

For example, let (C1, C2, C3, C4, C5, C6, C7, C8, C9, C10) be the cases of the base. We suppose that we can form the trios {(C1, C2, C3); (C2, C4, C5); (C6, C7, C8)}. We, thus obtain four segments. The first one comprises the trio (C1, C2, C3) and (C2, C4, C5), thus, the cases C1, C2, C3, C4 and C5. Thus, the cases C6, C7 and C8 will have their own segment as they do not share any case with the other segments. The cases C9 and C10 are not included in any randomization trio; thus, each of them will have its own segment.

In order to perform the partitioning, we use a variant of the k-medoids classification algorithm [28]. The k-medoids is a classical partitioning technique that splits a set of n objects into k clusters, each of which is represented by an element. In the classification we propose, k is initially unknown. The segmentation is as follows (Fig. 1):

- The knowledge base will be partitioned in an undefined number of segments.
- Each segment will have a delegate that is the most representative element in the segment.

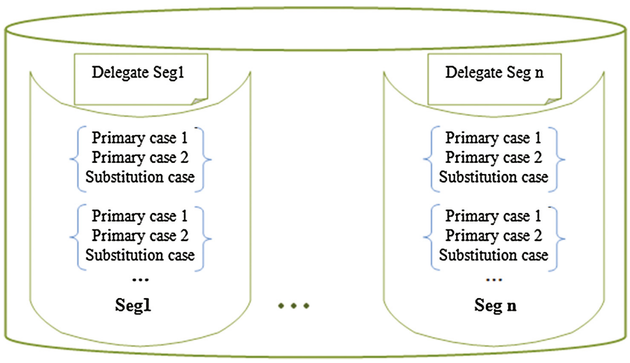


Fig. 1. Knowledge base structure.

- To be included in a segment, a new case is compared with the delegate; if they match, then the case is added; otherwise, it is compared with the other delegates.
- If a case does not match any delegate, it founds a new segment and becomes its delegate.

Figure 1 shows the structure of the knowledge base after segmentation.

We note that the cases are not physically inserted into the segments. They only are indexed. Thus, a case is saved once in the case base; but, it may be indexed several times in a segment.

While structuring the case base into segments, the latter are filled with the transmutation trios. Only trios generating valid cases are saved in the segments. Case validation is domain-specific. It is performed by domain experts. Furthermore, the inducted cases are not stored in the segments. This aims at preventing the explosion of the knowledge base. Thus, the transmutation process will be executed at every problem resolution as an alternative.

The case base segmentation is performed as given by Algorithm 1. No segment exists a priori.

Algorithm 1

```
For each case C
  For each segment S of the case base
    Compare the case C with the delegate of S
    If (comparison is true)
      Add the case to segment and choose
      Browse the segment and for each case C'
        If (C' is similar to C in either its problem
            or solution part)
              Form a pair of primary cases with C
              And C' and then save it
        End If
      End For
      Browse the segment a second time and for
        each case C''
          If (C'' is a "substitute case" for the
              Primary cases formed above)
                Save C'' with the associated primary
                cases (C and C') if a valid trio
          End If
      End For
    End If
  End For
  If no segment can contain C
      Create a new segment S'
      Add C to S' and define it as its delegate.
  End If
End For
```

4.2 Search for Accurate Knowledge

Figure 2 shows a CBR system integrating the proposed knowledge induction technique, which consists in the trios transmutation.

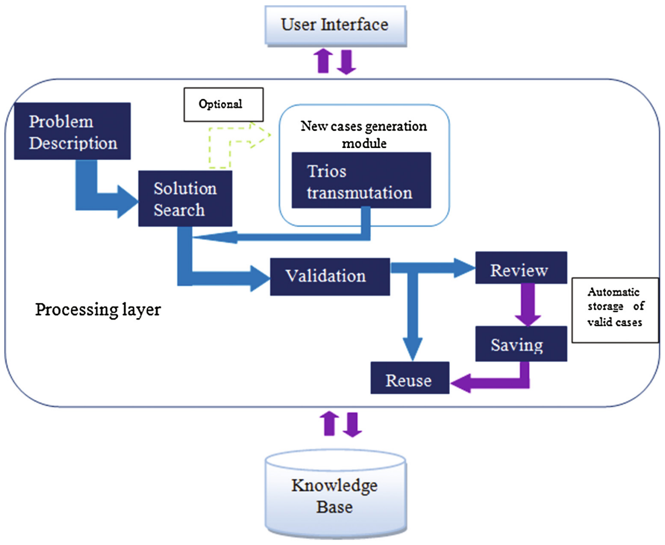


Fig. 2. CBR system architecture integrating knowledge induction

The user interface allows a user to specify the problem part of the searched case. Once the problem resolution is completed, it restores the solution part for the user. It also permits the system to interact with the user by giving him more detailed information about the problem or showing him the problems resolution steps.

The processing layer is based on the classical case-based reasoning process augmented with the knowledge induction process. The case-based reasoning process consists of 3 main phases: (1) description of the problem, (2) search of a solution among the cases relevant to solve the problem, and (3) reuse of a potential solution.

At the search phase, for a given problem resolution, a comparison is first made with the segment delegates. Only the segments whose delegate matches the problem to solve are retained. Next, prior to any search for the most similar case, we first generate more cases by running the transmutation on the trios of the retained segments. This may help finding a solution from the generated cases. It at least improves the chances for obtaining more similar cases than the ones previously entered into the knowledge base. Next, similarity is calculated between the problem to be solved and all cases of the retained segments. Search is given by Algorithm 2.

Algorithm 2

```
For each segment S of the case base
    Compare the case C with the delegate of S
    If (comparison is true)
        Add cases in S to the set SetOfCases
        For each trio T in S
            Generate new cases by transmutation on T
            Add the generated cases to SetOfCases
        End For

    End If
End For
Apply similarity measurement between the
problem of case C and all cases in SetOfCases
```

The reuse phase permits one to build the solution based on the most similar case found in the knowledge base. The more the cases that are similar, the simpler reuse will be, since the modifications will be relatively few. When similarity among the cases is not complete, there is need for adapting the retrieved solution so that it best fits the corresponding problem. Thus, validity of the retrieved solution is checked first. Next, the solution is possibly revised and the new case is saved to be reused in future problem resolutions.

We note that while the inducted cases, resulting from the transmutation process, are not saved in the case base, the revised cases are. Indeed, while the inducted cases can be re-derived, the revised cases will be lost if not stored.

5 Illustration of the Transmutation Approach

In order to best highlight the approach proposed in this paper, an appropriate domain of application is required. The application must have an intuitive solution in order to help to properly evaluate the approach. The degree of similarity between our intuitive knowledge about the results and the results obtained will reflect upon the validity of the approach.

As stated in the introductory section, we will apply our approach to a travelling robot that moves from point A to point B by going through a number of checkpoints. The problem to resolve will be the route described by the user. It consists of the start and end points plus all of the required checkpoints. The solution part must be a route including all checkpoints specified by the user. It is represented by a set of lines: Vertical Line (VL), Horizontal Line (HL) and Diagonal Line (DL).

Thus, each case will be represented as follows:

(Problem Situation) → (Solution Action); where:
Problem Situation: (start point (S), {checkpoints R1, R2, … Rn}, end point (F))
Solution Action: ({HL(S,A), VL (A,B), DL (B,R1), …,})

The segment delegates must be designed to best represent the segment cases. Thus, they should contain the pertinent points allowing for a good compromise between the number of segments and their size. A delegate composed of the start and end points as well as the checkpoints will provide a huge number of segments; whereas a delegate composed of only one point, for example the start point, will generate a large segment size. As a compromise, we suggest to compose the delegate of the start and end points.

In what follows, we note prb_i and sol_i, respectively, the problem and solution parts of $Case_i$.

5.1 Domain Analysis

Trios construction relies on similarity between the cases, especially those composing pairs of primary cases; whereas similarity depends on the domain we consider, i.e. the problem/solution semantics. An adequate semantics-based similarity function is usually more significant than a syntactic-based one, yielding the sought analogy between cases. A syntactic evaluation, through an integral comparison, may not provide accurate results. Herein, analogy depends on the chosen problem/solution granularity, expressed in terms of points to go through.

More specifically, in the retained domain, the problem part provides the situation, i.e. the points that must be crossed. Given the nature of the application, it means that at least the mentioned points must be visited. This is a hard constraint that must be satisfied by the solution; whereas, the solution action may be released by including more lines. We can intuitively understand that a route contains other points than those specifically indicated. As a result, we can affirm that any route (solution action) including all points mentioned in the problem is a solution for this problem. On the other hand, any situation whose points form concatenated stretches of a route (solution action) may be a problem for this solution. However, in this case, observance of strong sequencing is required.

A substitution case must include, within its problem (resp. solution) part, the problem (resp. solution) part of one case from the primary pair. Inclusion is released with regard to sequencing outside the checkpoints for the problem and solution parts. Points/lines may be added before and/or after the checkpoints, but never inside these latter.

In what follows, we will explore and discuss in detail how a new case is inferred.

5.2 Domain Semantics

Based on the domain analysis developed for the considered application, it follows that a precise semantics need be defined for the problem and solution parts. This semantics must support the necessary elements to express aspects related to route designation and route construction. Routes are traced by rectilinear movements. Hence, they respond to properties handling sequencing actions and/or states. We distinguish, hereafter, between three sequencing types.

Definition 1. Free sequencing
The free sequencing property is defined as a disordered sequencing of the elements of a set. It is expressed by means of the structural operator ";" to separate the elements of the set.

Definition 2. Weak sequencing
The weak sequencing property is defined as an ordered, but not necessarily joint sequencing of the elements of a set. Extra-set elements may be inserted between the elements of the set. This sequencing is expressed by means of the structural operator "|".

Definition 3. Strong sequencing
The strong sequencing property is defined as an ordered and joint sequencing of the elements of a set. None extra-set elements may be inserted between the elements of the set. This sequencing is expressed by means of the structural operator "||".

Definition 4. Sequencing substitution
Free sequencing substitution, weak sequencing substitution and strong sequencing substitution are substitutions where the substitute respectively satisfies free sequencing, weak sequencing, and strong sequencing.

Definition 5. Substitute
The substitute is the set of checkpoints, respectively lines, of a primary case, which replace the set of checkpoints, respectively lines, of the other primary case within a substitution case; e.g., in Sect. 3.1, the substitute is (A, E).

In terms of connection, the operator "||" is stronger than the operator "|" which is stronger than the operator ";".

For example, in S1 = (A; B; C | D | E || F || G), the sequence E||F||G is strongly connected, the sequence C|D|(E||F||G) may be separated by extra-set elements, and the elements A, B and (C|D|(E||F||G)) may appear in any order.

5.3 Transmutation Based on Problem Substitution

Let us consider the following cases:

Case1: (S1 || R1 || R2 || R3 || F1) → (HL(S1,R1), DL(R1,R2), DL(R2,R3), HL(R3, F1))

Case2: (S1 || **R1 | R3** || F1) → (HL(S1,R1), DL(R1,R2), DL(R2,R3), HL(R3,F1))

Case1 and Case2 compose a pair of primary cases that can be involved in transmutation by problem substitution as defined in Sect. 3.1. They have the same solution action for two different problem situations.

An appropriate substitution case would be Case3. Its problem situation includes all points specified by Case2. It is structured as follows.

Case3: (S1 || **R1 | R3** || R4 || R5 || F1) → (HL(S1,R1), DL(R1,R2), DL(R2,R3), HL (R3,R4), HL(R4,R5), HL(R5,F1))

Transmutation of the trio Case1, Case2 and Case3 yields the case Case3′.

Case3′: (S1 || R1 || R2 || R3 || R4 || R5 || F1) → (HL(S1,R1), DL(R1,R2), DL(R2, R3), HL(R3,R4), HL(R4,R5), HL(R5,F1))

Now, let us consider the following cases:

Case1: (S1 || **R1** || **R2** || **R3** || F1) → (HL(S1, R1), DL(R1, R2), DL(R2, R3), HL (R3, F1))
Case2′: (S1 | R1 | R3 | F1) → (HL(S1, R1), DL(R1, P), DL(P, R3), HL(R3, F1))

Case1 and Case2′ do present integral similarity neither for the problem nor for the solution parts. However, this does not mean that no similarity exists between the two cases. We notice that sol1 is a solution of prb1, but also a solution of prb2′. It provides a route that crosses all the points declared by prb2′, namely, the start and end points S1 and F1 as well as all the checkpoints (R1 and R3). Thus, prb2′ may be substituted for prb1. However, sol2′ does not resolve prb1. Thus, prb1 cannot be substituted for prb2′.

Thereby, Case1 and Case2′ compose a pair of primary cases, which can be involved in a transmutation by problem substitution. Case3 cannot constitute an appropriate substitution case for that pair of primary cases, since prb1 cannot be substituted for prb2′. Case4 does; its problem situation includes all points specified by prb1. It is structured as follows:

Case4: (S1 || **R1** || **R2** || **R3** || R4 || R5 || F1) → (HL(S1,R1), DL(R1,R2), DL(R2, R3), HL(R3,R4), HL(R4,R5), HL(R5,F1))

Transmutation of this new trio Case1, Case2′ and Case4 yields Case4′.

Case4′: (S1 || R1 | R3 || R4 || R5 || F1) → (HL(S1,R1), DL(R1,R2), DL(R2,R3), HL (R3,R4), HL(R4,R5), HL(R5,F1))

We note that R2 is no more a checkpoint in the generated case, but only a point included in the route.

5.4 Transmutation Based on Solution Substitution

Now, let us suppose that we have the following cases:

Case2: (S1 || R1 | R3 || F1) → (HL(S1, R1), **DL(R1, R2), DL(R2, R3)**, HL(R3, F1))
Case2′: (S1 || R1 | R3 || F1) → (HL(S1, R1), DL(R1, P), DL(P, R3), HL(R3, F1))

Case2 and Case2′ compose a pair of primary cases that can be involved in a transmutation by solution substitution as defined in Sect. 3.2. They have the same problem situation for two different solution actions.

An appropriate substitution case for this pair of primary cases would be Case5. Its solution action includes all stretches specified by Case2. It is structured as follows:

Case5: (S1 || R1 | R3 || R4 || F1) → (HL(S1, R1), **DL(R1, R2), DL(R2, R3),** HL (R3, R4), HL(R4, F1))

Transmutation of the trio Case2, Case2′ and Case5 yields the new case Case5′. It is equal to:

Case5′: (S1 || R1 | R3 || R4 || F1) → (HL(S1, R1), DL(R1, P), DL(P, R3), HL(R3, R4), HL(R4, F1))

Next, let us consider similarity between Case1 and Case2′ with regard to problem situations. We notice that both sol1 and sol2′ are solutions for prb2′. They both pass through R1 and R3 checkpoints and have the same start and end points S1 and F1. This implies that sol1 may be substituted for sol2′. However prb1 is not resolved by sol2′. Thus, sol2′ cannot be substituted for sol1.

Case1 and Case2′ thus compose a pair of primary cases, which can be involved in a transmutation by solution substitution. Case5 cannot be an appropriate substitution case for that pair of primary cases, since sol2′ cannot be substituted for sol1. But Case6 does; its solution action includes all lines specified by Case2′. It is given by:

Case6: (S1 || R1 | R3 || R4 || F1) → (HL(S1, R1), **DL(R1, P), DL(P, R3),** HL(R3, R4), HL(R4, F1))

Transmutation of this new trio Case1, Case2′ and Case6 yields Case6′.

Case6′: (S1 || R1 | R3 || R4 || F1) → (HL(S1, R1), **DL(R1, R2), DL(R2, R3),** HL (R3, R4), HL(R4, F1))

6 Validation of the Randomized Knowledge

6.1 Validation Rules

Consistency of the generated cases must be verified, since these latter cases have been generated using weak dependencies that cannot assure valid resulting cases. Hence, each generated case must be checked for solution consistency, problem consistency and case consistency. If consistency of a generated case is proved, then the latter can be used by the system. This means that its related trio is saved in the case base.

- **Solution consistency,** i.e. is the generated solution syntactically and semantically correct with regard to the domain of interest? For example, a non-continuous path from a source to a destination is an invalid solution.
- **Problem consistency,** i.e. does the generated problem contain an incoherence with regard to the domain of interest? For example, the problem has two start points.
- **Case consistency,** i.e. does the solution correspond to the associated problem? Does it achieve the goals targeted by the problem and satisfy its constraints? The problem and solution must have been previously validated.

6.2 Application to Strong Sequencing

Validation of the problem part consists in checking that: (i) the start and end points are the same as those of the delegate; (ii) substitute points are linked with the connector ||.

This is true with prb3′, since it includes the delegate's start and end points, namely S and F; and its sustitute points are linked with ||.

Validation of the solution part consists in checking that: (i) the first point of the first line is the start point of the delegate, and the last point of the last line is the end of the delegate; (ii) all lines are connected, thus forming a continuous route.

This is true with sol5′, since the first point of the first line is S, the start point of the delegate, and the last point of the last line is F, the end of the delegate; besides, all lines are connected, thus forming a continuous route.

Validation of a case consists in checking the following: (i) all points of the problem part must appear in the lines of the route composing the solution part; (ii) only those points must apear in the lines of the route; (iii) the order of these points must be kept the same in the lines composing the route as that of the problem part.

This is true with Case3′ and Case5′, since all points of their problem part appear in the lines of their routes; only those points apear in the lines of the routes; the order of these points is kept the same in the lines composing the route as that of the problem part.

6.3 Application to Weak Sequencing

Validation of the problem part consists in checking that: (i) the start and end points are the same as those of the delegate; (ii) all points are linked with the connector |.

This is true with prb4′, since it includes the delegate's start and end points, namely S and F; and its points are linked with |.

Validation of the solution part consists in checking that: (i) the first point of the first line is the start point of the delegate, and the last point of the last line is the end of the delegate; (ii) all lines are connected, thus forming a continuous route.

This is true with sol6′, since the first point of the first line is S, the start point of the delegate, and the last point of the last line is F, the end of the delegate; besides, all lines are connected, thus forming a continuous route.

Validation of a case consists in checking the following: (i) all points of the problem part must appear in the lines of the route composing the solution part.

This is true with Case4′ and Case6′, since all points of their problem part appear in the lines of their routes; and the order of these points is kept the same in the lines composing the route as that of the problem part.

6.4 Application to Free Sequencing

Validation of the problem part consists in checking that: (i) the start and end points are the same as those of the delegate; (ii) all points are linked with the connector ";".

This situation does not appear in our application.

Validation of the solution part consists in checking that: (i) the first point of the first line is the start point of the delegate, and the last point of the last line is the end of the delegate; (ii) all lines are connected, thus forming a continuous route.

Validation of a case consists in checking the following: (i) all points of the problem part must appear in the lines of the route composing the solution part; (ii) the order of these points may be different in the lines composing the route from that of the problem part.

7 Transmutation in Case Based-Reasoning

In this section, we will show how randomization is applied to case-based reasoning. It should increase the probability to retrieve the searched case by improving retrieval of new problems and/or retrieval of the pertinent solutions. We will show how knowledge induction either by problem transmutation and solution transmutation benefit, each one on its side, to case-based reasoning.

7.1 Problem Transmutation in CBR

Knowledge generation by problem transmutation provides new cases with novel problems and their corresponding solutions. We are, for example, interested in searching a route including the following points: (S1||R1||R2||R3||R4||R5||F1). This succession of points composes the problem part of the case, which we note prb. To retrieve prb in the case base, we proceed as follows.

First, the segment whose delegate is given by S1-F1 is retrieved. We note that the problem prb does not match with any problem part of the cases belonging to the segment. Transmutation is performed on every trio within the segment to generate new cases with a problem that would match the searched problem prb. Especially, the following trio provides a new case case3′, whose problem matches prb:

Case1: (S1 **|| R1 || R2 || R3 ||** F1) → (HL(S1,R1), DL(R1,R2), DL(R2,R3), HL(R3, F1))
Case2: (S1 || R1 | R3 || F1) → (HL(S1,R1), DL(R1,R2), DL(R2,R3), HL(R3,F1))
Case3: (S1 **|| R1 | R3 ||** R4 || R5 || F1) → (HL(S1,R1), DL(R1,R2), DL(R2,R3), HL (R3,R4), HL(R4,R5), HL(R5,F1))
Case3′: (S1 || R1 || R2 || R3 || R4 || R5 || F1) → (HL(S1,R1), DL(R1,R2), DL(R2, R3), HL(R3,R4), HL(R4,R5), HL(R5,F1))

Since the solution of case3′ has already been validated as a good solution in the phase of randomization, it will be retained for the searched problem prb.

7.2 Solution Transmutation in CBR

Knowledge induction by solution transmutation cannot help retrieving a searched problem, as problem transmutation does. Rather, it may help improving retrieval of the good solution by providing more than one solution for the searched problem. The user will be offered to choose the most convenient one to his problem.

We are, for example, interested in searching a route including the following points: (S1||R1|R3||R4||F1). This problem search involves a solution transmutation on the following trio:

Case2: (S1 ‖ R1 | R3 ‖ F1) → (HL(S1,R1), DL(R1,R2), DL(R2,R3), HL(R3,F1))
Case2′: (S1 ‖ R1 | R3 ‖ F1) → (HL(S1,R1), **DL(R1,P), DL(P,R3),** HL(R3,F1))
Case5: (S1 ‖ R1 | R3 ‖ R4 ‖ F1) → (HL(S1,R1), **DL(R1,R2), DL(R2,R3),** HL(R3, R4), HL(R4,F1))

This results in a new case5′, whose problem is the same as that of case5; but, their solutions are different.

Case5′: (S1 ‖ R1 | R3 ‖ R4 ‖ F1) → (HL(S1,R1), DL(R1,P), DL(P,R3), HL(R3, R4), HL(R4,F1))

It follows that the user will be provided with a double solution for the searched problem.

8 Implementation

Figure 3 shows the user interface and the result for a transmutation by problem substitution of the trio Case7-Case8-Case9, thus inducting Case9'. Here, the comma repre-sents a weak sequencing. The trio is structured as follows:

Case7: (S1, R1, R2, R3, F1) → (VL(S1,R1), HL(R1,A), VL(A,B), HL(B,R2), VL (R2,C), HL(C,R3), VL(R3,D), HL(D,G), VL(G,H), HL(H,F1))
Case8: (S1, **R1, R3,** F1) → (VL(S1,R1), HL(R1,A), VL(A,B), HL(B,R2), VL(R2, C), HL(C,R3), VL(R3,D), HL(D,G), VL(G,H), HL(H,F1))
Case9: (S1, **R1, R3,** R4, R5, F1) → (VL(S1,R1), HL(R1,A), VL(A,B), HL(B,R2), VL(R2,C), HL(C,R3), VL(R3,D), HL(D,R4), DL(R4,E), HL(E,R5), VL(R5,F), HL (F,F1))
Case9′: (S1, R1, R2, R3, R4, R5, F1) → (VL(S1,R1), HL(R1,A), VL(A,B), HL(B, R2), VL(R2,C), HL(C,R3), VL(R3,D), HL(D,R4), DL(R4,E), HL(E,R5), VL(R5, F), HL(F,F1))

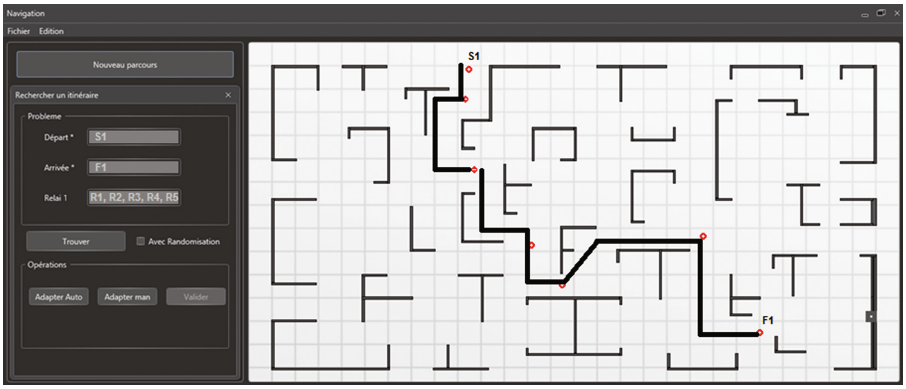

Départ * S1
Arrivée * F1
Relai 1 R1, R2, R3, R4, R5

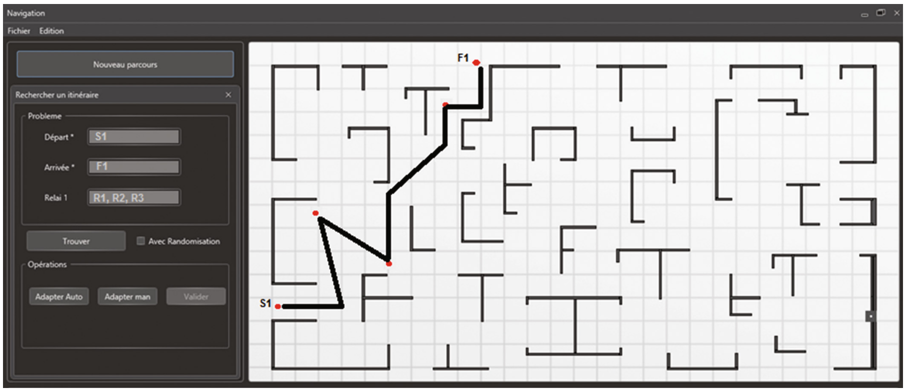
Fig. 3. Transmutation by problem substitution

Induction of Case9′ allows for providing a solution for the searched problem (S1, R1, R2, R3, R4, R5, F1). It generates a new case, with an additional checkpoint R2, in the path between S1 and F1, through the known checkpoints R1, R3, R4, R5.

Figure 4 shows the result for a transmutation by solution substitution of the trio Case10-Case11-Case12, thus inducting Case12′. The trio is structured as follows:

Case10: (S1, R1, R2, F1) → (HL(S1,A), DL(A,R1), DL(R1,R2), VL(R2,B), DL(B,F1), VL(R3,F1))
Case11: (S1, R1, R2, F1) → (HL(S1,A), DL(A,R1), DL(R1,R2), **VL(R2,C), DL(C,F1)**, VL(R3,F1))
Case12: (S1, R1, R2, R3, F1) → (HL(S1,A), DL(A,R1), DL(R1,R2), VL(R2,B), DL(B,D), VL(D,R3)), HL(R3,E), VL(E,F1))
Case12′: (S1, R1, R2, R3, F1) → (HL(S1,A), DL(A,R1), DL(R1,R2), **VL(R2,C), DL(C,D)**, VL(D,R3)), HL(R3,E), VL(E,F1))

Induction of Case10′ allows for providing a new route passing through the three checkpoints R1, R2 and R3.



Départ * S1
Arrivée * F1
Relai 1 R1, R2, R3

Fig. 4. Transmutation by solution substitution.

9 Conclusion

The approach proposed in this article allows for an enhancement of problem resolution in case-based reasoning systems by amplifying knowledge while saving space and speeding up search. Knowledge amplification is based on a technique of trios transmutation. Space saving is deduced from the key idea around case transmutation. Implicit knowledge within the case base is materialized, every time it is necessary - without the need to be saved. Only the relevant information, namely the valid transmutation trios, allowing for this materialization is saved. Knowledge materialization is obtained by transmutation, which scales well with CBR systems. Time gains are obtained through the proposed segmentation strategy based on delegates generalizing the problem description.

However, the inducted knowledge is not necessarily valid. It necessitates validation. This validation is domain- specific. We proposed to formalize the semantics of a domain specific application and validate the randomized knowledge.

References

1. Rubin, S.H.: Computing with words. IEEE Trans. Syst. Man Cybern. Part B **29**(4), 518–524 (1999)
2. Rubin, S.H., Murthy, S.N.J., Smith, M.H., Trajkovic, L.: KASER: knowledge amplification by structured expert randomization. IEEE Trans. Syst. Man Cybern. Part B Cybern. **34**(6), 2317–2329 (2004)
3. Rubin, S.H.: Case-Based Generalization (CBG) for Increasing the Applicability and Ease of Access to Case-Based Knowledge for Predicting COAs. NC No. 101366 (2011)
4. Rubin, S.H.: Multilevel Constraint-Based Randomization Adapting Case-Based Learning to Fuse Sensor Data for Autonomous Predictive Analysis. NC 101614 (2012)
5. Rubin, S.H., Bouabana-Tebibel, T.: NNCS: randomization and informed search for novel naval cyber strategies. In: Recent Advances in Computational Intelligence in Defense and Security. Studies in Computational Intelligence, vol. 621, pp. 193–223. Springer, Cham, December 2015
6. Rubin, S.H., Bouabana-Tebibel, T.: Naval intelligent authentication and support through randomization and transformative search. In: New Approaches in Intelligent Control and Image Analysis. Intelligent Systems Reference Library, vol. 107, pp. 73–108. Springer (2016)
7. Rubin, S.H.: Learning in the large: case-based software systems design. In: IEEE International Conference on Systems, Man, and Cybernetics, Decision Aiding for Complex Systems, Conference Proceedings (1991)
8. Bouabana-Tebibel, T., Rubin, S.H., Chebba, A., Bédiar, S., Iskounen, S.: Knowledge induction based on randomization in case-based reasoning. In: The 17th IEEE International Conference on Information Reuse and Integration – IEEE IRI 2016, Pittsburgh, USA, 28–30 July 2016
9. Begum, S., Ahmed, M.U., Funk, P., Xiong, N., Folke, M.: Case-based reasoning systems in the health sciences: a survey on recent trends and developments. IEEE Trans. Syst. Man Cybern. Part C Appl. Rev. **41**(4), 421–434 (2011)
10. Bridge, D., Healy, P.: GhostWriter-2.0 case-based reasoning system for making content suggestions to the authors of product reviews. Knowl. Based Syst. **29**, 93–103 (2012)
11. Wang, C., Yang, H.: A recommender mechanism based on case-based reasoning. Expert Syst. Appl. **39**(4), 4335–4343 (2012)
12. Hu, J., Guo, Y., Peng, Y.: A CBR system for injection mould design based on ontology: a case study. Comput. Aided Des. **44**, 496–508 (2012)
13. Yang, H.Z., Chen, J.F., Ma, N., Wang, D.Y.: Implementation of knowledge-based engineering methodology in ship structural design. Comput. Aided Des. **44**, 196–202 (2012)
14. Xie, X., Lin, L., Zhong, S.: Handling missing values and unmatched features in a CBR system for hydro-generator design. Comput. Aided Des. **45**, 963–976 (2013)
15. Ahn, H., Kim, K.: Global optimization of case-based reasoning for breast cytology diagnosis. Expert Syst. Appl. **36**, 724–734 (2009)

16. Naderpajouh, N., Afshar, A.: A case-based reasoning approach to application of value engineering methodology in the construction industry. Constr. Manag. Econ. **26**(4), 363–372 (2008)
17. Zarandi, F.M., Razaee, Z.S., Karbasian, M.: A fuzzy case based reasoning approach to value engineering. Expert Syst. Appl. **38**(8), 9334–9339 (2011)
18. Shi, L.-X., Peng, W.-L., Zhang, W.-N.: Research on reusable design information of shoes based on CBR. In: 2012 International Conference on Solid State Devices and Materials Science. Physics Procedia, vol. 25, pp. 2089–2095. Elsevier (2012)
19. Fan, Z.-P., Li, Y.-H., Wang, X., Liu, Y.: Hybrid similarity measure for case retrieval in CBR and its application to emergency response towards gas explosion. Expert Syst. Appl. **41**, 2526–2534 (2014). Elsevier
20. Khanuma, A., Muftib, M., Javeda, M.Y., Shafiqa, M.Z.: Fuzzy case-based reasoning for facial expression recognition. Fuzzy Sets Syst. **160**, 231–250 (2009)
21. Mittal, A., Sharma, K.K., Dalal, S.: Applying clustering algorithm in case retrieval phase of the case-based reasoning. Int. J. Res. Aspects Eng. Manag. **1**(2), 14–16 (2014)
22. Behbood, V., Lu, J., Zhang, G.: Fuzzy refinement domain adaptation for long term prediction in banking ecosystem. IEEE Trans. Ind. Inf. **10**(2), 1637–1646 (2014)
23. Shell, J., Coupland, S.: Fuzzy transfer learning: methodology and application. Inf. Sci. **293**, 59–79 (2015)
24. Li, S., Ho, H.: Predicting financial activity with evolutionary fuzzy case-based reasoning. Expert Syst. Appl. **36**(1), 411–422 (2009)
25. Sampaio, L.N., Tedesco, P.C.A.R., Monteiro, J.A.S., Cunha, P.R.F.: A knowledge and collaboration-based CBR process to improve network performance-related support activities. Expert Syst. Appl. **41**, 5466–5482 (2014)
26. Lao, S.I., Choy, K.L., Ho, G.T.S., Yam, R.C.M., Tsim, Y.C., Poon, T.C.: Achieving quality assurance functionality in the food industry using a hybrid case-based reasoning and fuzzy logic approach. Expert Syst. Appl. **39**, 5251–5261 (2012)
27. Kumar, Y.J., Salim, N., Abuobieda, A., Albaham, A.T.: Multi document summarization based on news components using fuzzy cross-document relations. Appl. Soft Comput. **21**, 265–279 (2014)
28. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review. ACM Comput. Surv. (CSUR) **31**, 264–323 (1999)

Utilizing Semantic Techniques for Automatic Code Reuse in Software Repositories

Awny Alnusair[1(✉)], Majdi Rawashdeh[2], M. Anwar Hossain[3], and Mohammed F. Alhamid[3]

[1] Department of Informatics and CS, Indiana University, Kokomo, IN 46904, USA
alnusair@iuk.edu

[2] Department of MIS, Princess Sumaya University for Technology, Amman 11941, Jordan
m.rawashdeh@psut.edu.jo

[3] Department of Software Engineering, King Saud University, Riyadh 11543, Saudi Arabia
{mahossain,mohalhamid}@ksu.edu.sa

Abstract. Reusing large software systems, including libraries of reusable components, is often time consuming. While some core library features can be well-documented, most other features lack informative API documentation that can help developers locate needed components. Furthermore, even when needed components are found, such libraries provide little help in the form of code examples that show how to instantiate these components. This article proposes a system that provides two features to help combat these difficulties. Firstly, we provide a mechanism for retrieving reusable components. Secondly, we provide support for generating context-sensitive code snippets that demonstrate how the retrieved components can be instantiated and reused in the programmer's current project. This approach utilizes semantic modeling and ontology formalisms in order to conceptualize and reverse-engineer the hidden knowledge in library code. Empirical evaluation demonstrates that semantic techniques are efficient and can improve programmer's productivity when retrieving components and recommending code samples.

Keywords: Component retrieval · Code recommendation · Software reuse · Semantic inference · Ontology models

1 Introduction

Code reuse provides many benefits during the development and maintenance of large-scale software systems. Nowadays, their exist a myriad of object-oriented reuse libraries. However, the ability to understand and reuse these libraries remains a challenging problem. The size and complexity of these libraries often inflict a significant burden on developers as they have to simultaneously grapple with multiple formidable learning curves during the reuse process. On one hand,

S.H. Rubin and T. Bouabana-Tebibel (eds.), *Quality Software Through Reuse and Integration*, Advances in Intelligent Systems and Computing 561,
DOI 10.1007/978-3-319-56157-8_3

Application Programming Interfaces (APIs) provided by libraries are not intuitive and often lack a well-defined structure and rich documentation. Thus, complicating the process of discovering and searching for reusable components. On the other hand, most libraries are not backed by enough source-code examples that would explain their features and the services they provide to potential reusers.

In this chapter, we propose an approach that assists developers in reducing these burdens. In order to demonstrate the utility of our approach, we have designed and implemented an integrated semantic-based system that can be used for component search and discovery. In order to further maximize the reuse potential, this tool can also be used to automatically construct personalized code examples that would show the proper usage of library components. As such, the tool is quite comprehensive as it provides two functionalities that complement each other. Thus, reducing the efforts of switching between tools and helping users stay focused on the current task.

To illustrate the system's capabilities, consider a developer who is trying to reuse Apache Jena [1], a Java open-source framework for building Semantic Web and Linked Data applications. In this scenario, the developer wishes to programmatically obtain an Ontology Class view of a Resource identified by a given Uniform Resource Identifier (URI). This view can be used to determine if an Ontology Class with the same URI exists in the Ontology Model. The developer would invoke our tool to find a Jena component that represents the ontological property (i.e. `Property`) and another component that represents the ontology class (i.e. `OntClass`). Users can use the tool to search for components by describing their functionality, signatures, or any role they play in the library. However, finding these components provides a partial solution because the next natural step would require the developer to write a code snippet that transforms an object of type `Property` into `OntClass`. A programming task of this kind can be seen as an object instantiation task of the form ($Property \longmapsto OntClass$). The following code snippet shows a sample solution for this query:

```
Property pr = ..... ;
Model model = pr.getModel();
OntModel ontModel = (OntModel) model;
OntClass ontClass = ontModel.getOntClass(pr.getURI());
```

Accomplishing this conceptually simple task should not be time consuming. However, due to the need of type casting and some intermediate steps, writing this short code snippet can be problematic, especially for novice programmers. Even worse, some other snippets would require complex call sequences for getting a handle of an object by invoking static methods from other hidden classes. Nevertheless, the proposed system provides a facility for retrieving the needed components as well as automatically constructing the code snippet.

In order to tackle our two interconnected program understanding and reuse features that are supported by our system, an effective approach must guide the construction of a mental model needed by software engineers in performing their daily development or maintenance activities [2]. Building a mental model that promotes software comprehension and captures the structure and organization of software systems requires a formal, explicit, and semantic-based representation of the conceptual knowledge of source-code artifacts provided by these systems. This article explores this hypothesis by providing an ontology-based representation of software knowledge.

Due to its solid formal and reasoning foundation, an ontology provides a data model that explicitly describes concepts, objects, properties and other entities in a given application domain. It also captures the relationships that hold among these ontological entities. Our previous work [3] explores the idea of using ontologies to retrieve software components and recommend code snippets. The work presented in this article, however, delves into the details and extends the previous work with updated ontologies and new experiments and case studies for testing both features supported by our system combined.

The rest of this chapter is organized as follows: In the following section, we position this approach within the context of current state of the art in code recommendation and component retrieval. In Sect. 3, we introduce the concepts, ontology languages, and Semantic Web technologies that are used in this approach. Section 4 discusses the details of the proposed mechanism for component identification and retrieval. In particular, we provide a component search scenario that explains how a reasoning system can operate on the semantic knowledge representation of software components in order to achieve better precision when retrieving components. Similarly, Sect. 5 discusses the essence of our approach for code recommendation. Specifically, we use a recommendation scenario to show how we can formulate a recommendation problem as query of the form *Source* $\longmapsto$*Destination*, and how our tool answers such query by utilizing the graph-based semantic representation of a software system. In Sect. 6, we present and discuss the results of two experiments that we have conducted to evaluate the proposed approach. In particular, we present the results of a component and code search experiment and another human experiment we conducted in controlled settings. Finally, we conclude in Sect. 7.

2 Related Work

There is a significant body of research related to software reuse; much of this has been implemented into useful tools. However, we could not find a proposal that combines component search and code recommendation in one unified environment. Therefore, we discuss these efforts separately.

2.1 Example Code Recommendation

Many recommendation approaches are based on analyzing a large corpus of sample client code collected either via Google Code Search (PARSEWeb [6]), or by searching in a pre-populated local repository (Satsy [7], Strathcona [8], Prospector [9], and XSnippet [10]). Strathcona for example uses heuristics to match the structure of the code under development to the structure of the code in the repository. Satsy is a Java application that uses symbolic execution to generate a repository by indexing a large collection of programs and stores them in a MySQL database. PARSEWeb, Prospector, and XSnippet, on the other hand, are more focused on answering specific object instantiation queries.

Our approach differs in several ways, specifically when dealing with issues related data gathering, data processing, and most importantly, knowledge representation. Firstly, with the exception of Prospector, other approaches rely on a repository populated with client code that expresses good usages of the framework. Prospector does analyze API signatures, but it relies on such repository to handle downcasts. Handling downcasts in our system as well as CodeHint [11] is done by evaluating code at runtime and hence can improve search accuracy by precisely finding the dynamic type of method calls. Effectiveness of the approaches that rely on repositories can be improved when a larger corpus of good samples of client code is analyzed.

2.2 Component Retrieval

Many component retrieval approaches such as [12] use traditional knowledge representation mechanisms with either signature matching or keyword-based retrieval. Similar to the proposed tool, other research tools (e.g., [13,14]) have been designed to be embedded in the user's development environment. However, these tools rely on the existence of a repository of sample code to perform the search. For example, CodeBroker [13] uses a combination of free-text and signature matching techniques. In order to retrieve appropriate matches, the user writes comments that precisely describe component's functionality. If comments did not retrieve satisfactory results, the system considers the signature of the method immediately following the comments.

Sugumaran and Storey [15] proposed an approach that utilizes a domain ontology used primarily for term disambiguation and query refinement of keyword-based queries; these keywords are then mapped against the ontology to ensure that correct terms are being used. Other proposals ([16] and [17]) employ ontologies to address the knowledge representation problem found in previous approaches. In [17], software assets are classified into domain categories (I/O, Security, etc.) and indexed with a domain field as well as other book-keeping fields to facilitate free text search. Although SRS [16] uses similar indexing mechanism, it maintains two separate ontologies; an ontology for describing software assets and a domain ontology for classifying these assets. However, the structure of source-code assets and the semantic relationships between these assets via axioms and role restrictions were not fully utilized.

3 Ontology-Based Software Representation

In this section, we introduce the concepts, ontology languages, and Semantic Web technologies that are used in this proposal. We further show how we utilize these technologies to build a structure of ontology models that serves as a basis for building a semantic-rich Knowledge Base (KB) that facilitates code reuse.

The ontologies we use in this approach were authored using the Web Ontology Language (OWL-DL)[1]. OWL is a knowledge representation and modeling language that formally captures semantic relationships among domain concepts. OWL-DL is a subset of OWL that is based on Description Logic (DL) and therefore it provides maximum expressiveness while maintaining computational completeness and decidability. OWL-DL's reasoning support enables inferring additional knowledge and computing the classification hierarchy (subsumption reasoning).

Figure 1 shows the modular structure of our ontology models linked via the `owl:imports` mechanism, which imports contents from an ontology into the current one. SCRO is a OWL-DL source-code ontology that describes the source-code's structure. SCRO is the base ontology model for understanding the relationships and dependencies among source-code artifacts. It captures major concepts and features of object-oriented programs, including class and interface inheritance, method overloading and overriding, method signatures, and encapsulation mechanisms.

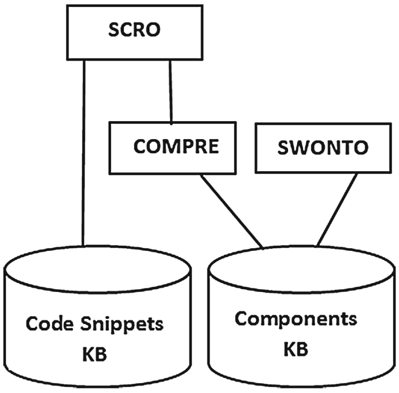


Fig. 1. Ontologies for code recommendation and component retrieval

SCRO defines various OWL classes and subclasses that map directly to source-code elements. It also defines OWL object properties, sub-properties, and ontological axioms that represent the various relationships among ontological concepts. SCRO is precise, well-documented, and freely available online [4], which allows it to be reused or extended by interested ontology reusers.

COMPRE is a component retrieval ontology. It inherits all definitions and axioms defined in SCRO. This ontology defines an API component as a distinct entity that is enriched with additional component-specific descriptions.

SWONTO is a domain specific ontology we developed in order to provide semantic annotations of component structures. Domain ontologies describe

[1] http://www.w3.org/TR/owl2-overview/.

domain concepts and their relationships concisely such that concepts are understandable by both computers and developers. COMPRE and SWONTO offer formalisms that collectively provide a semantic-based component retrieval mechanism that alleviates the current problems with signature-based and keyword-based approaches.

3.1 Knowledge Base Creation

After having these ontologies developed, we generate two distinct knowledge-bases that can be used as basis for automatic code recommendation and component retrieval as shown in Fig. 1. These KBs are generated by a subsystem we have developed for Java. This subsystem parses the Java bytecode and captures every ontology concept that represents a source-code element and automatically generates semantic instances for all ontological properties defined in SCRO for those program elements. The generated instances are serialized using the Resource Description Framework (RDF)[2], a language used to enable conceptual description of resources and provides an extensible model for representing machine-processable semantics of data. For each software library we parse, an RDF ontology that represents the instantiated KB for that library is created by our tool.

As we shall see in Sect. 4, capturing the metatdata modeled by some of COMPRE's datatype properties requires direct parsing of the framework's source-code. Therefore, we also capture and normalize identifiers, method signatures, comments, and Java annotations in order to obtain meaningful descriptions of software components. These descriptions are lexically analyzed and indexed using the tokenization and indexing mechanisms provided by Lucene[3], a high-performance full-featured text search engine.

In the following sections, we show how the generated KBs are used for component search and code recommendation. For viewing KB samples, we refer the reader to our ontologies website [4].

4 Component Retrieval

As discussed in Sect. 3 of this chapter, COMPRE captures component-specific descriptions and the interrelationships between library components. We further utilize domain-specific ontologies to provide semantic annotations of component structures. Domain-specific ontologies have been widely recognized as effective means for representing concepts and their relationships in a particular domain such as finance or medicine. SWONTO is a domain ontology that we have built for the "Semantic Web Applications Domain". It provides a common vocabulary with unambiguous and conceptually sound terms that can be used to annotate software components in this domain.

[2] http://www.w3.org/TR/rdf-primer.

[3] http://lucene.apache.org/.

To this end, we were able to generate an enhanced KB that describes library code as specified by SCRO and COMPRE and its application domain as specified by SWONTO. Semantic search for components is accomplished by matching user's requests expressed as terms from the domain ontology against component descriptions in the populated KB. As discussed in Sect. 3, the process of generating semantic instances for the concepts and relations specified in SCRO and COMPRE's datatype properties is completely automatic. However, the process of annotating components according to COMPRE's object properties is currently done manually as it is the case for semantic annotations in general. Our tool, however, provides a view that allows users to enter these annotations right into the KB.

4.1 Component Search Scenario

The components KB we discussed earlier provides multi-faceted descriptions of components, including their interfaces and their relationships with each other. Since this is an ontology-based KB, it enables semantic reasoning. As such, we utilized a sound OWL-DL reasoner to enrich our KB with additional logical consequences from the existing facts and axioms that are already stated in the KB. The result is a RDF-based KB that we can use to execute pure semantic-based queries.

RDF represents knowledge using a labeled directed graph that is made from the set of KB triples. Figure 2 shows a partial RDF description of the Jena's `read(..)` method, which reads a query from a file. Unlike traditional graph-representations, RDF graphs provide precise description of resources and they are capable of encoding metadata in their nodes (represented by OWL classes) and edges (represented by OWL object properties). As such, RDF graphs provide flexible means for semantic reasoning and deductive querying.

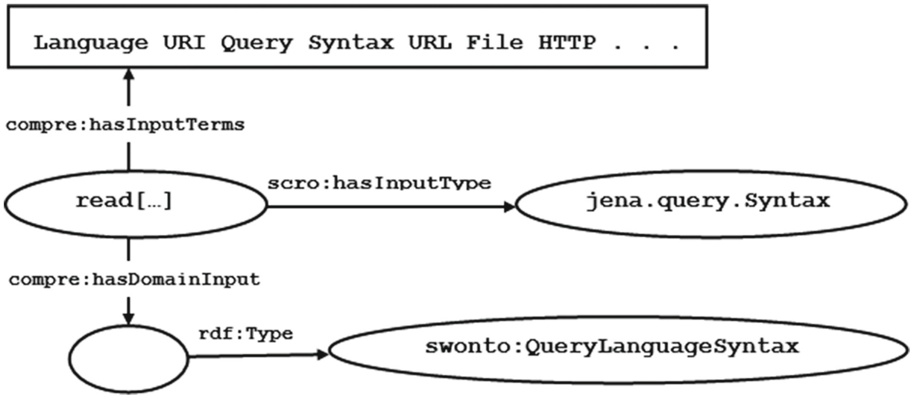


Fig. 2. Partial RDF multi-faceted graph representation of the Jena's read[..] method

The graph of the `read(..)` method uses the `hasInputType` SCRO property to assert that this method has an input parameter of type `Syntax`. The graph also shows descriptions that associates the method's input parameters with some terms that explain their purposes. This is done using the `hasInputTerms` OWL property from the COMPRE ontology. In order to complete the description of this method's inputs, we tag it with domain concepts from the SWONTO domain ontology using `swonto:QueryLanguageSyntax`. The full graph representation of this method is large to be shown here but it also associates this method with domain concepts that describe its purpose, its return type, and other component descriptions.

This representation of components overcomes known problems with keyword-based or type-based component search. It also allows us to execute different kinds of searches against the KB, including multi-faceted search. For example, let's search for a component with a domain input of type `ExtendedQuery` and one of its input types is `Syntax`. Additionally, let's assume that we are not sure about other input types so we provide few terms as filters to limit the recommendations. This search scenario can be expressed using the following query in DL-like syntax:

$$\begin{aligned}
&Query \equiv compre : Component \sqcap \\
&(\exists compre : hasDomainOutput\ .\ \ swonto : ExtendedQuery) \sqcap \\
&(\exists scro : hasInputType\ .\ kb : Syntax) \sqcap \\
&(\exists compre : hasInputTerms\ value\ "url\ \ file")
\end{aligned}$$

A large number of components can be retrieved when executing some searches against the KB. Therefore, our system uses efficient ranking heuristics to reorder the results. We initially rely on Lucene's scoring mechanisms. We further improve this ranking based on context-sensitive measures. In this regard, we use context parameters based on parsing and analyzing the code in the user's project - especially visible data types that are currently declared by the user and those that appear in the retrieved component's signatures. As such, recommended items that do not introduce additional types are ranked higher. These simple heuristics are easy to implement and work surprisingly well.

5 Automatic Code Recommendation

Most libraries are shipped with few code examples that are not enough to establish a solid basis for comprehension and reuse. Our system tries to overcome this problem by providing a recommendation mechanism for constructing relevant code examples based on user's requests. We particularly emphasize the automaticity, relevancy, and accuracy of the recommended code examples.

As outlined in Sect. 1, our approach for code recommendation focuses on answering user queries of the form *Source* $\longmapsto$*Destination*. In the special case where the *Source* object is not stated, the object instantiation problem is reduced to either a simple constructor call or a static method invocation. The procedure we use to recommend code uses ontology formalisms to describe software assets by building a RDF knowledge base that conforms to the SCRO ontology. We thus construct a code example by traversing the RDF graph that represent the framework in question. The RDF graph that we use for code recommendation is similar to the one we presented in Sect. 4.1. However, since this graph is based on the code-example KB, we use only representations from our SCRO ontology.

5.1 Code Recommendation Scenario

Consider a developer who wishes to programmatically execute a Jena SELECT query. The developer typically starts with a `String` reference representing the query and wishes to obtain a `ResultSet` object representing the solution. For a developer who is unfamiliar with Jena, accomplishing this task may not be easy because there are some intermediate steps and various method invocations for instantiating the desired `ResultSet` object. These steps are outlined in the following code snippet:

```
String queryString ="...";
Query query = QueryFactory.create(queryString);
QueryExecution qe = QueryExecutionFactory.create(query);
ResultSet rs = qe.executeSelect();
```

Our system recommends such code example by viewing it as an object-instantiation task, which shows how to get a handle of the `ResultSet` reference. This is accomplished using the semantic representations and RDF graph representation.

Figure 3 shows a partial RDF graph of the code example described above. The `hasOutputType` is a SCRO functional property that represents the method's return type. `hasInputType` represents the type of a method's formal parameter.

Given this directed RDF graph, we construct the needed code snippet by performing brute-force traversal starting at the node that represents the source type (`Query`) and systematically enumerating all possible paths to the destination type (`ResultSet`). For each path candidate, we automatically generates a code snippet and present it to the user.

Generating a code snippet in this manner may produce a large number of potential candidates. Therefore, we use heuristics to rank the results based on relevancy and user's context. In particular, we use the path size heuristic and user context heuristics. The path size represents the number of RDF statements that are necessary to compose the code example. This heuristic assigns top rank to the shortest path in the graph. On the other hand, similar to component search, context-based heuristics assigns higher ranks to paths that better fit within the user context in terms of the code that has already been developed by the user. These two heuristics are simple, yet helped improve the ranking of code examples.

5.2 Dealing with Downcasts

Some complex libraries do not usually provide type-specific methods that relief the developer from having to use downcasts. Unlike other approaches that handle

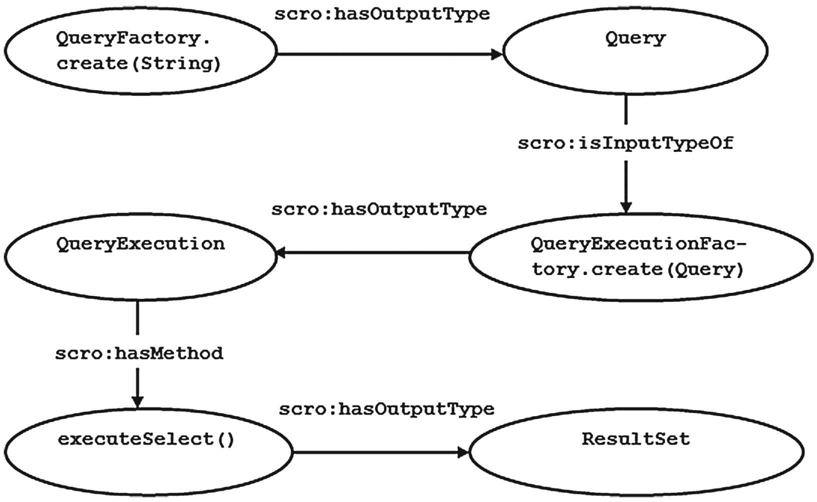


Fig. 3. Subset of the RDF graph for constructing: *Query* $\longmapsto$*ResultSet*

downcasts by analyzing a corpus of carefully selected client code, we instead rely on the `hasActualOutputType` OWL property from SCRO. This property represents the actual runtime return type of a given method. Therefore, we enrich the graph with an edge labeled with `hasActualOutputType` that leaves a given method and enters every possible runtime type of this method. When we construct a code example, `hasActualOutputType` is treated as an expression casting the result of the method down to its actual return type. Obtaining the precise return type of methods is accomplished using interprocedural points-to and call graph analysis techniques [5], a set of static program analysis techniques that analyze a given program in order to obtain precise reference and call-target information.

Suggesting a code snippet with a downcast is not the norm. In fact, the need to downcast reveals hidden complexities in the underlying framework. Consider a programmer coding for the Eclipse API who wishes to obtain a handle of `JavaInspectExpression`, which represents the currently selected expression in the debugger from an object of type `IDebugView`. The code below details a sample solution:

```
IDebugView debugger = ...
Viewer viewer = debugger.getViewer();
IStructuredSelection sel =
                (IStructuredSelection) viewer.getSelection();
JavaInspectExpression expr =
                (JavaInspectExpression) sel.getFirstElement();
```

In order to construct such snippet, we rely on points-to-analysis. Figure 4(a) shows part of the RDF graph where the dead end is reached. `ISelection` has in fact a single method of no relevance, `isEmpty()`, and there is no way for the traversal algorithm to proceed further. We thus infer the runtime return type of method `getSelection()` by using flow-insensitive points-to analysis on the partially constructed snippet.

Upon completion, this analysis concludes that `IStructuredSelection` is the actual return type of `getSelection()` at that particular call site (dashed lines represent the `hasActualOutputType` property). The same process is repeated for `getFirstElement()` as shown in Fig. 4(b). As such, we avoided using a static corpus of client code that may in fact have no answer for this query.

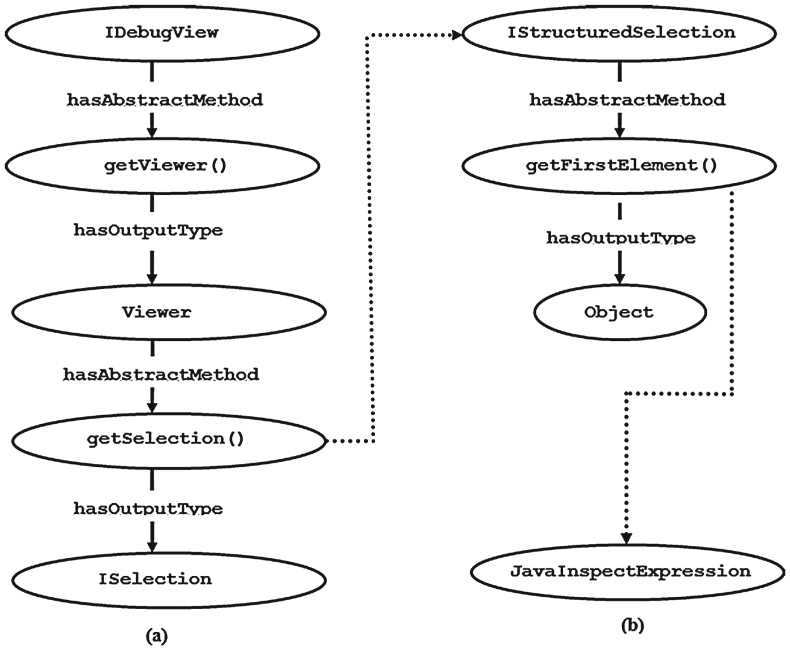


Fig. 4. *IDebugView* $\longmapsto$ *JavaInspectExpression*

6 Evaluation

Our tool was developed as a plugin for the Eclipse IDE and it is available online [4]. This tool is supported by radically optimized set of ontologies, extended parsing support, and enhanced reasoning and inference support. It is also supported by enhanced interprocedural points-to analysis measures using Soot[4], a program analysis and instrumentation framework commonly used to improve the performance of Java and AspectJ programs.

[4] https://sable.github.io/soot/.

When using this tool, users can formulate queries using Eclipse views that are designed as simple data entry forms. The component search view provides entry boxes for entering search restrictions and the code recommendation view provides an entry box for the source object and another for the destination object. Once the form is filled out, the tool automatically generates a query and executes it against the KB.

Although we have tested both of the features supported by our tool individually, the following subsections discuss two primary experiments that we have conducted to evaluate both features combined. We used the Jena framework in these experiments because the domain ontology described in Sect. 4 is created for the Jena's application domain.

6.1 Component and Code Search Experiment

This experiment is designed to ensure that our system helps programmers when they attempt to reuse software libraries. In particular, we hypothesized that ontology formalisms with reasoning support improve precision and programmer's productivity when searching for components and constructing valid code examples that effectively utilize these components. We have designed 10 Jena programming tasks and carefully selected another 10 from the Jena developers forums, newsgroups, and stackoverflow.com. Each task requires searching for two distinct components that solve an object instantiation problem. The descriptions of these tasks are not shown here for space constraints. However, based on our extensive experience with Jena, we believe that these tasks represent good samples of common Jena programming issues and would fully exercise our ontologies, including the domain ontology.

We instructed our system to parse the Jena library and create the RDF ontology as explained in Sect. 3.1. We then loaded the generated KB and the domain ontology into the plugin. This experiment was performed on a 2.4 GHz machine with 4 GB RAM running MS Windows 7.

Table 1 presents experiment results. **Precision** is defined as the ratio of the number of *relevant* component instances that are recommended by the system to the total number of recommended components.

On average, component search retrieved 3.4 matches when searching for two components, ranging from 1 (exact match) to 8 components as shown in PT18. In the last two tasks, PT19 and PT20, only one query was needed per task to search for the destination component. The system found and retrieved a single component in response to each query.

Consider PT18, the query for Op found 7 matches and Op itself was found at position 6. The query for the source component, QueryIterator, retrieved only one component and it was the correct match. The tool performed poorly when searching for Op because this component and its methods were not annotated properly with appropriate mappings between COMPRE's object properties to precise SWONTO domain concepts. In this experiment, component search with domain-aware semantic search performs well in terms of ranking and precision as it encodes the necessary information about each component's internal structure

Table 1. Experiment results: searching for components and constructing code examples

Programming tasks	Component retrieval (CR)			Code examples (CE)			Running time	
	Rank[1]	Num.[2]	Prec.[3]	Rank[4]	Size[5]	Total[6]	CR[7]	CE[7]
PT1: `EnhGraph` $\longmapsto$ `Profile`	4	7	0.29	2	6	13	16	9
PT2: `Model` $\longmapsto$ `Resource`	2	2	1	1	2	7	10	6
PT3: `Query` $\longmapsto$ `ResultSet`	2	2	1	2	5	12	12	5
PT4: `Model` $\longmapsto$ `Property`	3	3	0.67	1	3	12	12	5
PT5: `InfModel` $\longmapsto$ `Resource`	2	2	1	1	3	15	10	5
PT6: `OntModel` $\longmapsto$ `Profile`	3	3	0.67	1	2	20	6	8
PT7: `Graph` $\longmapsto$ `ModelCom`	4	4	0.5	1	2	20	6	9
PT8: `Model` $\longmapsto$ `InfGraph`	2	2	1	3	6	10	14	9
PT9: `ResultSet` $\longmapsto$ `Literal`	3	3	0.67	2	5	8	8	10
PT10: `Model` $\longmapsto$ `Individual`	2	2	1	1	5	20	10	10
PT11: `Query` $\longmapsto$ `Model`	2	2	1	1	4	18	10	6
PT12: `Dataset` $\longmapsto$ `QueryExecution`	2	2	1	5	6	20	6	12
PT13: `MyQueryEngine` $\longmapsto$ `Plan`	3	7	0.28	1	2	6	8	3
PT14: `QueryExecution` $\longmapsto$ `StageGenerator`	2	2	1	1	6	10	12	5
PT15: `QuerySolution` $\longmapsto$ `RDFNode`	2	2	1	1	2	20	10	10
PT16: `OntClass` $\longmapsto$ `Individual`	3	3	0.67	1	3	20	10	10
PT17: `BasicPattern` $\longmapsto$ `OpBGP`	4	6	0.34	1	3	20	12	13
PT18: `QueryIterator` $\longmapsto$ `Op`	7	8	0.25	5	7	15	16	9
PT19: `Null` $\longmapsto$ `DataSource`	1	1	1	1	2	20	3	4
PT20: `String` $\longmapsto$ `Model`	1	1	1	0	0	20	4	12
Averages	2.8	3.4	0.77	1.6	3.7	15.3	10.2	8

[1] Rank of desired components: combined value for both src and dest components
[2] Total number of retrieved components: combined value for both src and dest
[3] Precision value per component (2/Num.) or (1/Num.) in last two tasks
[4] Rank of desired code example if found or 0 if not found
[5] Path size: number of RDF statements in the sub-graph between src and dest
[6] Total number of generated code examples. The tool produces 20 at the most
[7] Time in seconds to search for both components (CR) and to construct the code example (CE)

and its precise relationships with other components. In most cases, the tool achieved good results in terms of ranking components due to its capabilities of incorporating the context in which the component was searched for.

The system achieved 0.77 precision value on average. This high rate is due to precise annotations of domain concepts in the domain ontology and due to the use of reasoning support. In situations where precision suffers, either the components were not annotated properly or the query was not precise enough to point out the proper description of the component. Regardless, the results we obtained were good enough to show the value of semantic annotations in component search, which in turns supports our hypothesis.

In terms of code recommendation, our system achieved good ranking scores. On average, the required snippet was among the top two. Once again, the system performed quite poorly in task PT18. This is because the algebra expression needed to be transformed using a reference of type `Transform`. Also, there were multiple references that must be included in the context to produce this quite long code example of 7 RDF statements. However, when the query was issued, there were only 2 of these references, `DataSetGraph` and `Binding`, included in the user's project. The code for PT12 was also ranked low. This is in part due to the fact that in order to execute the given query, a `Query` object was needed in addition to the source component, `Dataset`. This reference was not yet introduced in the user's project when the code example was requested.

Utilizing points-to analysis to resolve runtime type information was also assessed. Tasks PT1, PT6, PT10, and PT14 require casting. For these tasks, the tool ranked the required code example high, but in most cases the path size was considerably larger than other examples due to the need of traversing more graph nodes via the `hasActualOutputType` property. In PT10 for example, pointer analysis has determined that the actual type of the given `Model` reference was in fact `OntModel` and therefore it was safe to do the casting before invoking `createIndividual()` to create the needed anonymous individual reference.

Our system accepts *Null* $\longmapsto$*Destination* queries (e.g., PT19). These queries are quite useful when instantiating objects using a class constructor or a static method call, or when the source object is unknown. Since we rely on API signatures, the number of generated examples for these tasks can be large.

PT20 involves object instantiations with strings. In this task, the user starts with a String object representing a query and wishes to obtain a model object that represents the results of executing this query. Precision suffers when tasks require the highly polymorphic string objects. In this case, it was not clear to the tool that the string contains a query and therefore the query must be executed first via a Query object before being transformed to an ontology model. As such, the tool returns no valid code example among the first 20 recommendations. However, if the task was formulated as *Query* $\longmapsto$*Model*, the tool would have found the solution much more easily.

We have also timed our tool. As shown in the last two columns, on average, it took 5.1 s to retrieve a single component and 8 seconds to traverse the RDF graph and generate the required code example. Mostly, the time shown here is spent by the reasoning engine to process the KB and compute logical consequences from its statements and axioms. Due to recent advances of reasoners, the results shown here demonstrate a good improvement over similar experiments we have conducted in the past.

6.2 Human Experiment

In this section we report the results of a human experiment we conducted in controlled settings. The goal is to evaluate the proposed system's utility and performance, and how well potential users perform real-world tasks when reusing or maintaining an unfamiliar library. This experiment also intended to compare semantic code search with other search practices.

Participants

Our human subjects include 4 Java developers and 4 senior CS undergraduate students. All participants reported that they program on a daily basis and search for code quite frequently. On average, participants have had three years of Java experience, but no experience with the Jena framework. We divided participants into two groups. The *System Group* includes two randomly selected developers and two students. This group was allowed to use only our system to solve predefined programming tasks. The *Control Group* consists of the remaining four participants who were not allowed to use our tool, but they can use their own typical search practices.

Tasks and method
We have designed five Jena programming tasks. Each task requires a novice Jena user to search for source and destination components and write a code snippet that solves an object instantiation problem. The chosen tasks are consistent with the goals of this experiment. In particular, among the 10 tasks we have designed ourselves and used previously, we chose PT3, PT9, and PT12. Additionally, we added two other tasks, *QueryIterator* $\longmapsto$*Node* (PT21), and *QuerySolution* $\longmapsto$*Literal* (PT22). These tasks vary in scope and difficulty, would exercise the domain ontology, and casting was required.

We provided our subjects with a brief training session that introduces basic Semantic Web terminology, the domain ontology, and the Jena ARQ API. We also provided a sample training task that explains the nature of the experiment. For each task, we provided a working wrapper code that participants could use to develop their final solution. Finally, we asked each subject to record the components he/she identified for each task. This way we can track the percentage of component retrieval rate in case the participant has not fully solved the assigned problem.

The primary hypothesis we set out to test in this experiment is that the proposed system yields better outcome and shortens development time. In other words, the System Group will solve the given programming problems more quickly and more reliably. In order to test this hypothesis, we observed subjects and collected quantitative performance data, which include the time needed by each participant to either complete or abandon each task and the number of participants who successfully complete the tasks. Qualitative data was also collected in the form of a post-experiment questionnaire that was presented to the System Group. This 5-point Likert-scale questionnaire intended to find out how users feel about using the proposed system in terms of learnability, usefulness, and how helps programmers understand the recommended code. Furthermore, a semi-structured interview followed to learn more about the usability and value of this system.

Experiment results
In this experiment we tracked weather a participant fully completed the task, partially completed the task (i.e., identified the proper components but failed to write the code snippet) or not completed the task at all. Figure 5 shows an overview of the completion rates for each of the assigned five programming scenarios. For example, programmer P1 in the System Group has successfully completed all five tasks while programmer P2 in the same group has completed only PT3 and PT9, found the required components to complete PT12 and PT22, but was unable to write the code snippet, and he was unable to complete task PT21.

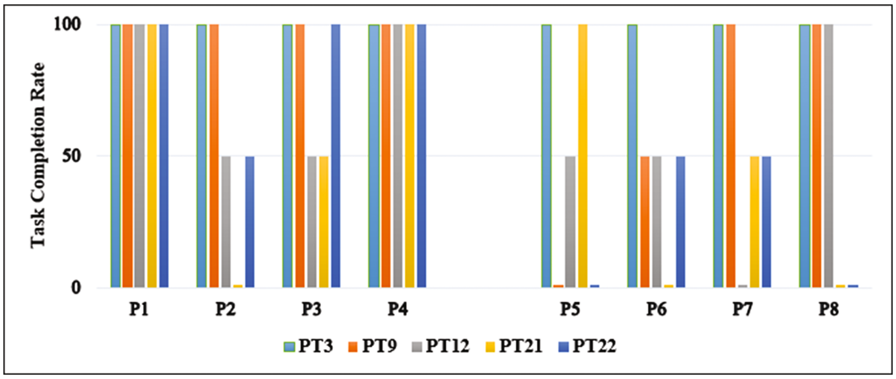


Fig. 5. User study: task completion data in the System Group (P1–P4) and Control Group (P5–P8). A task is either fully completed (100%), half completed (50%), or not completed at all (0%)

In the System Group, all participants were able to fully complete PT3 and PT9, only two programmers completed PT12 and PT21, and three programmers completed PT22. Thus, the System Group's overall completion rate for the entire task (searching for components and writing the code) is 75% and the search for components success rate is 95%. In the Control Group, however, the overall completion rate is 40% and the search success rate is 70%. Task PT3 was the first task programmers have to complete. This task was obtained from Jena documentation, which shows a standard way to obtain the results of executing a query and therefore all subjects have completed this task successfully.

It was relatively easy to find the components for PT12 except programmer P7, but only two programmers in the System Group and one programmer in the Control Group was able to write the code. This is due to the fact that a `Query` object was required in addition to the `Dataset` object to instantiate the destination object. This task in fact reveals a limitation of our approach. In particular, if the components have been more functionally related to each other, the snippet ranking would have been improved. This can be accomplished perhaps via semantic annotations or attaching a domain ontology to refine the choice and improve the ranking of the recommended code snippets.

PT21 and PT22 are examples of scenarios that require casting. 62% of programmers in the System Group were able to complete both tasks successfully, while only 12% (programmer P5 solved PT21) in the Control Group were successful. The remaining three programmers in the Control Group have arrived at bad casts that introduced more errors. This higher success rate in the System Group is achieved in part by the use of points-to analysis techniques that the system provides. As such, this experiment provided strong and conclusive support for the value of program analysis when coupled with semantic modeling in handling complexities in software libraries.

Figure 6 shows an overview of the time taken by each participant to complete each scenario. The time shown represents the overall time taken by each participant to successfully complete and test the solution for each API task. This includes searching for components and generating a valid code snippet that completes the program under construction. The graph shows average completion time for only those tasks that have been fully completed (i.e., times for incomplete or partially completed tasks were discarded).

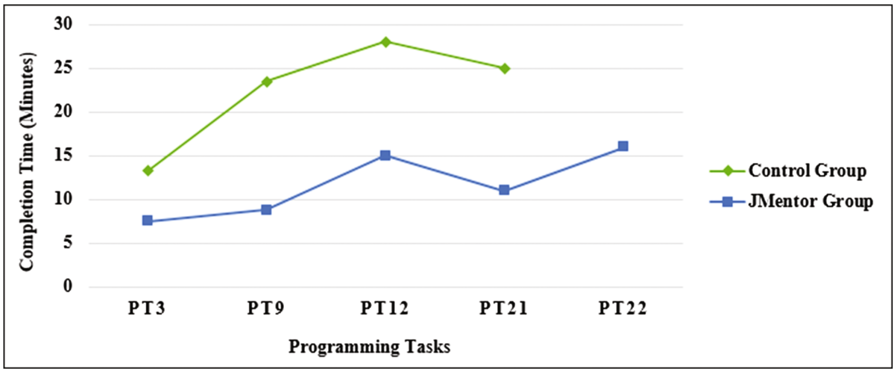


Fig. 6. User study: average task completion time in the system group and control group

In the vast majority of cases, programmers in the System Group fared better. As described above, PT12 was the most time consuming for participants. Tasks PT21 and PT22 took quite more time than others. The need for casting underlies a complication of any framework and therefore participants reported that they had to try multiple recommendations before finally selecting the correct code snippet. Participants also reported that the time to select the correct code snippet was minimal compared to the time needed to test the code. We believe that the system's ability to incorporate user's context in code search has helped shortening the development time.

Although the number of participants and the number of tasks used in this study are not very large to provide conclusive data with statistical evidence about development times, the averages shown in Fig. 6 suggest that the proposed system can help programmers to become more productive by completing their programming tasks faster.

Overall, this user study shows that the system can enhance programmer's learnability and efficiency when reusing unfamiliar frameworks. Most programmers in the System Group provided complimentary remarks regarding the system's usability and its techniques for delivering reusable components and recommending valid and concise code examples.

This study, however, did not provide decisive evidence that the use of the domain ontology helps programmers greatly in understanding the Jena domain. Two programmers argued that they have not given enough time to learn the taxonomy presented in the domain ontology, which otherwise could make them more productive in formulating more complex search queries during component search. Also, this study did not provide strong evidence that our system's ranking techniques are very efficient. Two subjects suggest that better ranking could have shortened the development time, especially in tasks PT12 and PT22 as they had to test multiple recommendations before arriving at the correct code that they could copy to complete the task. Three subjects agreed that in order to make the tool more usable, it should provide quick means in which programmers can insert snippet recommendations in the editor and test it out automatically in order to quickly filter out invalid snippets.

It was evident to us that while subjects in the Control Group were spending time searching documentations and the web for code examples, subjects in the System Group were more focused on the tasks by studying and understanding the tool's suggestions while gradually learning the Jena API. At the exit interview, control subjects indicated that a solid API should be equipped with enough code examples that explains its features and therefore a reliable recommendation system would make them more efficient in their daily programming activities.

6.3 Threats to Validity

There exists several external threats to the validity of these experiments. In particular, the validity of both experiments is limited by the choice and number of the software libraries used. Firstly, unlike most other frameworks, Jena is well-documented. Secondly, although we have obtained similar results by informally experimenting with other frameworks, including the JDBC library, the GEF plugin for Eclipse, and Apache Ant, these two experiments reported results about only selected Jena packages. Performing a more comprehensive evaluations with other libraries may further strengthen our conclusions and reduce some of these threats. Furthermore, although we have used frameworks of varying sizes and complexities, we are unable to establish the scalability of the reasoner's performance without testing it on more large-scale libraries.

Ontology development is usually subjective and therefore whoever creates the domain ontology is not the end user of our tool. As such, it would be interesting to experiment with other domain ontologies created by others. SWONTO is precise and reliable, but we are yet to experiment with other domain ontologies with lesser depth and breadth.

Finally, although the human experiment provided good insight about the tool's usability, this experiment is limited by the number of programmers in the System Group and the number of tasks used. A larger scale user study may give us better insight on how easy to learn the domain ontology. Ideally, we would like to have the same group of programmers experimenting with tasks that use our tool and other tasks that use typical search procedures. However, it was quite difficult to come up with tasks that have exact complexity levels. Furthermore, the order in which tasks are completed is quite significant as we expect users gain more experience with a given API with every task they complete. If such user experience evaluation is possible, it would certainly mitigate many threats.

Finally, since user studies involve human subjects, they tend to introduce an internal threat as programmers display varying degrees of proficiency in programming techniques. Another study that involves considerably larger number of programmers can certainly reduce this threat.

7 Conclusion

The notion of *Program Understanding* as described in this paper is quite a broad one. It encompasses the many processes and activities undertaken by software engineers to build effective mental models during the development of new software systems as well as maintaining and evolving existing ones. Similarly, this paper addressed the broad aspect of the notion of *Software Reuse*. Current reuse practices range from copying and pasting simple code fragments to utilizing complex domain-specific libraries when building or evolving large-scale software systems. A systematic approach to reuse is, however, encompasses the whole lifecycle of the development process and it is indeed essential to any domain-specific reuse practice. To this end, this paper proposed a semantic-based solution that exploits the formal and explicit semantics provided by ontologies to code recommendation and software components retrieval. This work uniquely combines these two areas in one environment that is based on semantic representations of software knowledge.

Our approach for automatic code recommendation assists programmers who try to reuse an unfamiliar framework by constructing and delivering personalized code examples. Our research tool creates code candidates, ranks them based on the user's context, and delivers these code examples for immediate insertion into the user's current project. In constructing such examples, the tool combines a RDF graph-based representation of a framework's classes and methods, augmented with information about potential targets of polymorphic calls, to produce the needed code example for object instantiation queries. It also employs interprocedural points-to and call graph analysis techniques to resolve run-time type information, which leads to proper handling of special Java features (e.g., type cast legality). This approach neither requires a carefully crafted corpus of sample code to mine for examples, nor it requires a source-code search engine to obtain these samples. Furthermore, precision is improved due to the underlying RDF graph representation. On the other hand, ranking of the recommended candidates is improved because of the context sensitive measures that have been used.

An approach for software component search has also been proposed in this paper. This approach supports different kinds of search mechanisms: users can perform pure semantic-based search, keyword-based search, and signature-based search. However, we focused the discussion on pure semantic search because based on our experiments, this mechanism tends to be the most precise of all supported search mechanisms. This semantic search relies on a vocabulary specified in a domain-specific ontology that can be used for tagging the components in the repository with unambiguous term descriptions of the component's functionality. Precision improvement and flexibility in formulating and refining user queries are the main strengths of this approach.

The work presented in this paper uniquely contributes to the proper linking of the established field of information integration and reuse and the emerging field of semantic web by developing a set of ontologies for the problems that programmers face when reusing software libraries. Our ontology models are precise, continuously updated, exhibit a certain depth and breadth that makes them good candidates to be reused in solving several other related problems. This is also true for the knowledge population parser that we have developed and continue to enhance so it can capture even hidden aspects of software dependencies and higher levels of source-code analysis in object-oriented libraries.

The work presented in this paper opens new doors for further enhancements and new research perspectives. Investigating the application of semantic annotations and domain ontologies to provide more guidance during code example construction as well as refining the choice among candidates is foreseen as a promising new direction. We have not yet investigated how could one motivate library providers to ship domain ontologies with their software. Such ontologies can certainly be created as a community effort. Therefore, a systematic way of creating and sharing these ontologies among a community of users would be an interesting future work. Ranking reuse candidates has been always a challenge for both code recommendation and component retrieval. It would be interesting to investigate the potential of using complex approximation techniques that are based on, for example, Natural Language Processing or Machine Learning, to improve ranking by capturing complex user patterns. Finally, we are currently studying the possibility of enhancing search by utilizing Collaborative Filtering and social tagging. These techniques emphasize community collaboration in order to improve search and artifact recommendation.

References

1. Apache Jena: An open-source framework for building semantic web and linked data applications. https://jena.apache.org/. Accessed 13 Oct 2016
2. Rilling, J., Meng, W.J., Witte, R., et al.: Story driven approach to software evolution. IET Softw. **2**(4), 304–320 (2008)
3. Alnusair, A., Rawashdeh, M., Alhamid, M.F., Hossain, M.A., Muhammad, G.: Reusing software libraries using semantic graphs. In: 17th IEEE International Conference on Information Reuse and Integration (IRI-2016), pp. 531–540. IEEE Press (2016)
4. Ontologies and the tool Website. http://www.indiana.edu/~awny/index.php/research/ontologies. Accessed 5 Nov 2016
5. Emami, M., Rakesh, G., Hendren, L.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 242–256 (1994)
6. Thummalapenta, S., Xie, T.: PARSEWeb: a programmer assistant for reusing open source code on the web. In: IEEE/ACM Conference on Automated Software Engineering, pp. 204–213 (2007)
7. Stolee, K.T., Elbaum, S., Dwyer, M.B.: Code search with input/output queries: generalizing, ranking, and assessment. J. Syst. Softw. **116**, 35–48 (2015)

8. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: International Conference on Software Engineering (ICSE), pp. 117–125 (2015)
9. Mandelin, D., Xu, L., Bodik, L., et al.: Jungloid mining: helping to navigate the API jungle. In: Conference on Programming Language Design and Implementation (PLDI), pp. 48–61 (2005)
10. Tansalarak, N., Claypool, K.: XSnippet: mining for sample code. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 413–430 (2006)
11. Galenson, J., Reames, P., Bodik, R., et al.: CodeHint: dynamic and interactive synthesis of code snippets. In: International Conference on Software Engineering, pp. 653–663 (2014)
12. Bajracharya, S., Ossher, O., Lopes, C.: Sourcerer: an internet-scale software repository. In: Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation (2009)
13. Ye, Y., Fischer, G.: Reuse-conductive development environments. Int. J. Autom. Softw. Eng. **12**(2), 199–235 (2005)
14. Shatnawi, A., Seriai, A.D.: Mining reusable software components from object oriented source code of a set of similar software. In: IEEE International Conference on Information Reuse and Integration, pp. 193–200 (2013)
15. Sugumaran, V., Storey, V.C.: A semantic-based approach to component retrieval. ACM SIGMIS Database **34**(3), 8–24 (2003)
16. Antunes, B., Gomez, P., Seco, N.: SRS: a software reuse system based on the semantic web. In: IEEE International Workshop on Semantic Web Enabled Software Engineering (2007)
17. Durao, F.A., Vanderlei, T.A., Almeida, E.S., et al.: Applying a semantic layer in a source code search tool. In: ACM Symposium on Applied Computing, pp. 1151–1157 (2008)

A Multi-strategy Approach for Ontology Reuse Through Matching and Integration Techniques

Enrico G. Caldarola[1,2] and Antonio M. Rinaldi[1,3](✉)

[1] Department of Electrical Engineering and Information Technologies, University of Naples, Federico II, Napoli, Italy
{enricogiacinto.caldarola,antoniomaria.rinaldi}@unina.it
[2] Institute of Industrial Technologies and Automation, National Research Council, Bari, Italy
[3] IKNOS-LAB Intelligent and Knowledge Systems, LUPT - University of Naples, Federico II, Napoli, Italy

Abstract. The new revolutionary web today, the *Semantic Web*, has augmented the previous one by promoting common data formats and exchange protocols in order to provide a framework that allows data to be shared and reused across application, enterprise, and community boundaries. This revolution, together with the increasing digitization of the world, has led to a high availability of *knowledge models*, i.e., more or less formal representations of concepts underlying a certain universe of discourse, which span throughout a wide range of topics, fields of study and applications, mostly heterogeneous from each other at a different dimensions. As more and more outbreaks of this new revolution light up, a major challenge came soon into sight: addressing the main objectives of the semantic web, the sharing and reuse of data, demands effective and efficient methodologies to mediate between models speaking different languages. Since ontologies are the *de facto* standard in representing and sharing knowledge models over the web, this paper presents a comprehensive methodology to ontology integration and reuse based on various matching techniques. The approach proposed here is supported by an ad hoc software framework whose scope is easing the creation of new ontologies by promoting the reuse of existing ones and automatizing, as much as possible, the whole ontology construction procedure.

Keywords: Ontology · Ontology integration · Ontology matching · Ontology reuse · Semantic network · WordNet

1 Introduction

Throughout the last decade, a more revolutionary web has emerged just when the main ideas and concepts behind the Web 2.0 were starting to enter into the main stream. The new web (the Semantic Web) has augmented the previous one by promoting common data formats and exchange protocols in order to provide a common framework that allows data to be shared and reused across application, enterprise, and community boundaries [50]. In this scenario, a new term has rapidly become a buzzword among the computer scientists, i.e., *Ontology*. Originally introduced by Aristotle, an ontology is a *formal specification of a shared conceptualization of a domain* [21], i.e., a formal definition and representation of the concepts and their relations belonging to a certain domain of interest. Once a knowledge domain or some aspects of it are formally represented using a common and shared language, they become understandable not only by humans but also by automated computer agents [7]. As a result, for example, web services or search engines can improve their performances in terms of exchange of information or accuracy in searching results, exploiting the semantically enriched representation of the information they share. For these reasons, ontologies are increasingly considered also as a key factor for enabling interoperability across heterogeneous systems [9], improve precision in retrieval process [38] and enhance efficient and effectiveness of document management and representation [39,40]. This revolution has led companies of all sizes and research groups to produce a plethora of data or conceptual models for many applications such as: e-commerce, government intelligence, medicine, manufacturing, etc. This, together with the increasing digitization of the world, has made available a huge amount of disparate information, raising the problem of managing heterogeneity among various information sources [46]. One of the most challenge consists in integrating heterogeneous models in a unified (homogeneous) conceptualization of a specific knowledge or application domain, which allows the reuse of existing knowledge models. This is not an easy task to face due to ambiguities, inconsistencies and heterogeneities, at different levels, that could stand in the way. The ability to effectively and efficiently perform knowledge reuse is a crucial factor in knowledge management systems, and it also represents a potential solution to the problem of standardization of information and a viaticum towards the realization of the Semantic web. Furthermore, available ontologies in the literature are becoming increasingly large in terms of number of concepts and relations to such an extent that technical solutions belonging to the Big Data landscape can be adopted in order to make scalable ontology operations like storage, visualization and matching [5,6]. One of the most notable applications of ontology integration is the *ontology reuse*. In the context of ontology engineering, reuse of existing knowledge models is recommended as a key factor to develop cost effective and high quality ontologies, since it reduces the cost and the time required for creating ontologies *ex novo* increasing the quality of newly implemented ones by reusing components that have already been validated [1,3]. Finally, it avoids the confusion and the inconsistencies that may be generated from multiple representations of the same domain; thus, it strengthens the orchestration and harmonization of knowledge.

Taking into account the above considerations and acknowledging that ontologies are the de facto standard in representing and sharing knowledge models over the web, this paper presents a multi-strategy methodology to ontology integration and reuse, based on different matching techniques. Starting from a literature review of the main existing approaches and methodologies, we will show how the adoption of a multi-strategy approach based on existing techniques and on an *ad hoc* software framework can improve and simplify the creation of new ontology models, automatizing as much as possible the ontology creation task and promoting the knowledge reuse. Although the proposed approach tries to reduce the human intervention in all ontology integration phases, does not neglect it, neither considers it as en element of weakness; on the contrary, user involvement is considered an essential *tuner* for the whole strategy as it incorporates precious user knowledge in the framework making it more effective. The proposed approach will be applied to the food domain, specifically the food production domain, by collecting and subsequently analysing some of the most spread knowledge models available in the literature. Nevertheless, the approach does not loose generality and can be applied to other knowledge domains. What described here extends a previous work by the authors [4] as follows:

– more explanations have been added for the process of normalization and adaptation of heterogeneous reference knowledge models by describing the types of different model formats and the strategies used to homogenize such formats;
– the matching techniques within the matching phase have been also detailed by describing how the *matcher* acts in order to obtain an effective alignment between the reference models and the target model;
– the integration phase is also detailed describing the strategy used to merge equivalent concepts and construct the *is-a* hierarchy of concepts starting from the integrated input ontologies;
– a more complete characterization of the reference models is given by providing some metrics referred to the main models;
– some considerations are added regarding the scalability of the whole framework w.r.t. the volume of reference models outlining the strategies that can be used to face this issue.

The reminder of the paper is structured as follows. After a clarification of the terminology related to the discussed research areas provided in the next subsection, Sect. 2 reviews the main ontology integration and reuse methodologies existing in the literature. A description of the proposed approach is provided in Sect. 3 and a detailed outline of the matching methodologies is provided in Sect. 4. Section 5 applies the entire approach to a specific case study, the food production, highlighting results and strengths. Finally, Sect. 6 shows the experimental results in terms of efficiency and effectiveness of the entire procedure while the last section draws the conclusion summarizing the major findings and outlining future investigations.

1.1 Ontology Matching and Integration Background

Welcoming the suggestion for a clarification of the terminology contained in [17], we provide here some definitions about the key concepts used in this work in order to establish a solid background for the successive sections. According to some related works in the literature [15,35], we define *ontology matching* as the process of finding relationships or correspondences between entities of different ontologies; *ontology alignment*, as a set of correspondences between two or more ontologies; *ontology mapping*, as the oriented, or directed, version of an alignment, i.e., it maps the entities of one ontology to at most one entity of another ontology. More formally, the ontology mapping can be defined according to [24] as the task of relating the vocabulary of two ontologies that share the same domain of discourse in such a way that the ontological signatures and their intended interpretations, as specified by the ontological axioms, are respected. *Ontology integration* and *merging* are defined as the construction of a new ontology based on the information found in two or more source ontologies; and finally, *ontology reuse* as the process in which available ontologies are used as input to generate new ontologies. Less used but with a broader meaning is the term *ontology change* [17], which refers to any type of modification that it is needed over an ontology in response to particular needs. Its sense includes changes due to heterogeneity issues, ontology engineering updates, ontology maintenance, etc.

2 Related Works

It is a common practice in the literature to consider heterogeneity resolution and related ontology matching or mapping strategies to be an internal part of ontology merging or integration [22]. Several works have been addressed in the last decade to ameliorate the ontology mapping and matching strategies for an effective and efficient data integration. According to Choi [9], ontology mapping can be classified into three categories: (1) mapping between an integrated global ontology and local ontologies, (2) mapping between local ontologies and (3) mapping on ontology merging and alignment. The first category of ontology mapping supports ontology integration by investigating the relationship between an integrated global ontology and local ontologies. The second category enables interoperability by providing a mediation layer to collate local ontologies distributed between different nodes. The third category is used as a part of ontology merging or alignment in an ontology reuse process. Some of the most spread tools belonging to this category will be described as follows. SMART [33] is an algorithm that provides a semi-automatic approach to ontology merging and alignment assisting the ontology developer by performing certain tasks. It looks for linguistically similar class names through class-name matches, creates a list of initial linguistic similarity (synonym, shared substring, common suffix, and common prefix) based on class-name similarity, studies the structures of relation in merged concepts, and matches slot names and slot value types. SMART also determines possible inconsistencies in the state of the ontology that may result from the user's actions, and suggests ways to remedy these inconsistencies. Another semi-automatic ontology merging and alignment tool is PROMPT [34]. This performs some tasks automatically and guides the user in performing other tasks for which his intervention is required. It is based on an general knowledge model and therefore can be applied across various platforms. Anchor-PROMPT [32] takes a set of anchors (pairs of related terms) from the source ontologies and traverses the paths between the anchors in the source ontologies. It compares the terms along these paths to identify similar terms and generates a set of new pairs of semantically similar terms. OntoMorph [8] provides a rule language for specifying mappings, and facilitates ontology merging and the generation of knowledge-base translators. It combines two powerful mechanisms for knowledge-base transformations: syntactic rewriting and semantic rewriting. The first one is done through pattern-directed rewrite rules for sentence-level transformation based on pattern matching, while the latter is done through semantic models and logical inference; FCA-Merge [48], which is a method for ontology merging based on Ganter and Wille's formal concept analysis, lattice exploration, and instances of ontologies to be merged; and finally, CHIMAERA [29], which is an interactive merging tool based on Ontolingual ontology editor. It makes users affect merging process at any point during merge process, analyzes ontologies to be merged, and if linguistic matches are found, the merge is processed automatically, otherwise, further actions can be made by the user. It uses subclass and super class relationship. A survey of the matching systems is also provided by Shvaiko and Euzenat in [46], where, in addition to an analytical comparison of the recent tools and techniques, the authors argue on the opportunity to pursue further researches in ontology matching and propose a list of promising directions for the future. Particularly, some recent trends and future challenges suggested by the authors are: large-scale knowledge bases integration and mediation and ontology matching using knowledge background [31]. The first challenge is also subject of interesting studies conducted by Wiederhold [51], who has defined the services (or functions) a domain-specific mediator module must guarantee in order to collect and mediate information coming from increasing large-scale information systems. It is worth investigating the second challenge that consists in matching two ontologies by discovering a common context or background knowledge for them and use it to extract relations between ontologies entities. Adding context can help to increase the recall but at the same time may also generate incorrect matches decreasing the precision, thus, a right tradeoff must be found. As background knowledge, it is common to use generic knowledge sources and tools, such as WordNet, Linked Open Data (LOD) like DBpedia, or the web itself, when the ontologies to be matched are common sense knowledge model, i.e., non specialistic knowledge bases. On the contrary, if they are low-level ontologies focused on a particular field of studies, domain specific ontologies can be used as background knowledge. The semantic matching framework S-Match [20], for example, uses WordNet as a linguistic oracle, while the work in [45] discusses the use of UMLS (Unified Medical Language System), instead of WordNet, as a knowledge background in medical applications.

2.1 Ontology Reuse

As mentioned in the introductory section, ontology integration is mainly applied when the main concern is the *reuse* of ontologies. In this regard, it is worth to note that several knowledge management methodologies consider the reuse of knowledge as an important phase of the entire knowledge management process. Common KADS methodology [43], for instance, makes use of a collection of ready-made model elements (a kind of building blocks) which prevent the knowledge engineer to *reinventing the wheel* when modeling a knowledge domain. Moreover, the European research project NeOn [49] proposed a novel methodology for building ontologies, which emphasizes the role of existing ontological and non-ontological resources for the knowledge reuse. Reuse is also a key requirement of OBO Foundry ontology [47], a collaborative effort to establish a set of principles for ontology development with the eventual goal of creating a set of interoperable reference ontologies in the domain of biomedicine [19]. The goal is to ensure that ontology developers reuse term definitions that others have already created rather than create their own definitions, thereby making the ontologies orthogonal, which means that each term is defined in only one ontology. Some recent works in the literature, mainly in the life sciences domain, still consider reuse as an important aspect of ontology construction or generation. OntoFox [53] is a web-based system that provides a timely publicly available service with different options for users to collect terms from external ontologies, making them available for reuse by import into client OWL ontologies. In [25] a semi-automatic ontology development methodology is proposed to ease the reusing phase in the development process, while [44] proposes a guiding framework for Ontology Reuse in the biomedical domain and [18] shows an approach to extract relevant ontology concepts and their relationships from a knowledge base of heterogeneous text documents. From a methodological point of view, Pinto and Martins [35] have analysed the process of knowledge reuse by introducing an approach that comprises several phases and activities. In particular they identify three meanings of ontology integration: when building a new ontology by reusing (assembling, extending, specialising or adapting) other ontologies already available; when building an ontology by merging several ontologies into a single one that unifies all of them; when building an application using one or more ontologies [24]. However, some open issues remain, especially concerning the difficulty of dealing with the extreme formalisms heterogeneity of the increasing number of models available in the literature. The absence of an automatic framework for the rigorous evaluation of the knowledge sources is also a severe limitation to overcome.

The proposed methodology distances the approaches described above since it represents a comprehensive framework for ontology integration and reuse, not focused on a specific task. In fact, it helps the developer from the early phases of the methodology (knowledge sources and domain identification) by suggesting guidelines, to the final, more specialized and technical phases, by providing a software framework for automatically accomplishing the matching and integration tasks.

3 The Proposed Framework for Ontology Integration and Reuse

This section describes a high-level architecture of the proposed framework for ontology integration and reuse. As shown in Fig. 1, our framework presents four main functional blocks, which, starting from the identification of the existing knowledge models in the literature (hereafter referred to as *reference models*), along with the reconciliation and normalization of such models, obtain a comprehensive and integrated representation of the domain under study, by an effective and efficient reuse of selected reference models existing in the literature.

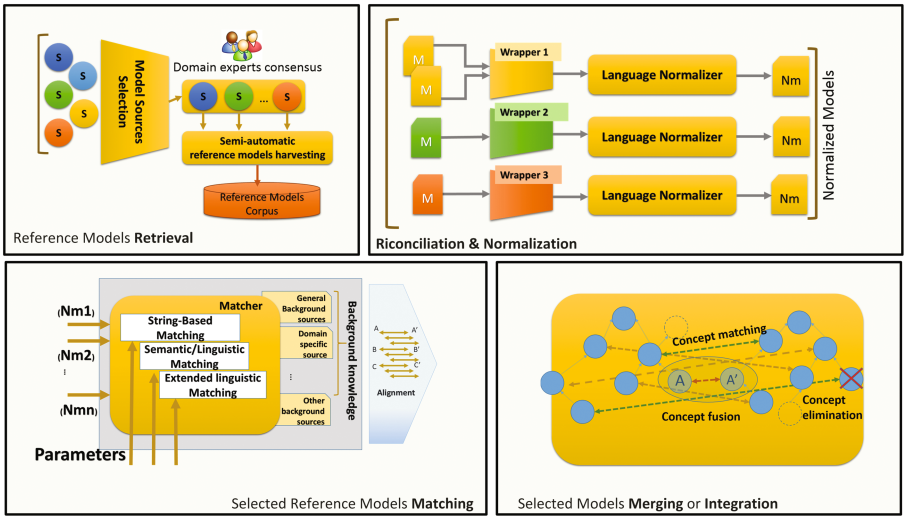


Fig. 1. High-level view of the proposed framework

3.1 Reference Models Retrieval

The first component of the framework is the Reference Models Retrieval function block. This model is responsible for retrieving the reference models corresponding to the domain of interest. In order to search for proper reference models, it is needed to identify the knowledge domain and the related sub-domains covering the specific topic under study. The contribution of users with domain expertise is essential in this phase. They clarify the meaning of some poorly defined concepts and help knowledge engineers moving among the existing knowledge sources over the Internet or other legacy archives. Some of the available resources for domain identification are:

– Wordnet [16], a freely and publicly available large lexical database of English words;

- General purpose or content-specific encyclopedia, e.g., Wikipedia and The Oxford Encyclopedia of Food and Drink in America;
- Web directories, e.g., DMOZ (from directory.mozilla.org) and Yahoo! Directory;
- Standard classifications, e.g., the International Classification for Standards (ICS) compiled by ISO (International Standardization Organization);
- Other electronic and hard-copy knowledge sources, including technical manuals, reports and any other documentation that the domain experts may consider useful to identify the knowledge domains.

Once the domain of interest has been properly defined, a corpus of selected reference models is populated by collecting them manually or using *ad hoc* search services from different knowledge sources:

- Specialized portals and websites within public or private organizations;
- Search engines (e.g., Google, Bing, etc.), directory-based engines, e.g., Yahoo!, BOTW (Best of the Web Directory), DMOZ, etc., specialized semantic-based engines, e.g., Yummly (specialized on food), True Knowledge, etc.;
- Ontology repositories including: BioPortal, Cupboard, Schemapedia, Knoodl, etc., and search engines for semantic web ontologies, e.g., Swoogle and the Watson Semantic search engine;
- Available standards and non-standard reference models. The former are created by standardization organizations like ISO, while the latter are created by private or public research groups without being internationally accepted reference standards. In this source category can be included knowledge model providing requirements, specifications, guidelines and characteristics of a service or a product (ISO standards, the IFC Industry Foundation Classes, Ansi/ISA-95, STEP, mentioned above, and the Core Product Model).

To make a first screening of reference models, a set of qualitative criteria can be adopted:

- *Language formality* (C1) which describes the formality of the conceptual model representation that can range from plain text with no formalism to formal languages like the first-order logic-based languages;
- *Domain specificity* (C2) which evaluates the model type from the viewpoint of its generality (upper-level or abstract domain model or application specific model);
- *Model structuring* (C3) which evaluates the model type from the viewpoint of its structure (simple classifications or taxonomies versus representation language based model like UML and EXPRESS);
- *Model language* (C4) which describes the language used to represent the conceptual model, including RDF/OWL (Resource Description Framework/Ontology Web Language), graphic-based languages and pure text;
- *Model provenance* (C5) which evaluates the model from the viewpoint of its origin, thus giving higher rates to standards or conceptual models authored by influential scientific groups. Finally,

– *Model availability* (C6) which evaluates the availability of the conceptual model (open data-model versus proprietary and licensed models).

A higher rate will be given to formal models written in OWL/RDF language because this is the format in which all input models will be converted through the adaptation stage. Concerning the domain specificity, it is preferable to avoid highly abstract or too specific models. For example, the application of this methodology to the case study described in the Sect. 5, i.e., the *Food production*, has led us to consider only ontologies with an average level of granularity for the concepts definition. We have discarded ontologies which have only few and generic concepts and ontologies which are linked to a specific application or a specific disciplinary sector (e.g., biochemistry, biomedical, etc.). Finally, only publicly available models have been taken into consideration and/or model provided by standardization organizations.

3.2 Reference Model Reconciliation and Normalization

The second component of the framework is the format and syntax adapter. It is responsible for adapting the collected reference models to a common representation format. This step is mandatory due to the heterogeneity of languages used to represent and formalize existing reference models in the literature. These can be represented using plain text, semi-structured text (like XML, EXPRESS, etc.), graphical languages (like UML, (E-R) Entity-Relationships models, etc.) or ontology languages with different levels of expressiveness (RDF, OWL-Lite, OWL-DL, OWL-Full, etc.). The idea behind this component is to create an adpater for each reference model retrieved from the Internet that access the model and transform it in a simple ontology, preserving the *is-a* hierarchy between the classes and all available linguistic annotations (labels and comments). Each adapter creates an OWL-Lite ontology in a memory based model using the Jena APIs [28]. This model use a transitive reasoner that allows transitive inference in the model itself. In general, if an ontological model (serialized in RDF or OWL language) is available as source model, it is imported in a Java class that loads it inside a Jena ontology model. In other cases, if the source model has an high volume of classes, the adapter modules acts as a kind of crawler searching for ontology fragments specifically regarding the domain under study. Once the fragments have been scraped from the source model, they are transformed in a OWL

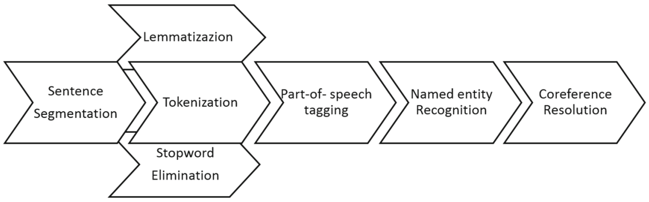


Fig. 2. Text processing pipeline in the normalization phase

Lite model and serialized in a OWL file like for the previous described cases. If source models are expressed through UML or other graphical-based modelling languages, the transformation from the former models to an OWL model can be automatized at a meta-model level. In [54], authors analyse the similarities between UML and OWL, investigating transformation rules from one model to the other using the QVT (Query/View/Transformation) language.

Once the reference models have been converted in proper adapted OWL ontologies, they go through the text-processing pipeline. Figure 2 shows the main phases of the pipeline:

– *Sentence Segmentation* phase is responsible for breaking up documents (entity description, comments or abstract) into sentences (or sentence-like) objects which can be processed and annotated by "downstream" components;
– *Tokenization* breaks sentences into sets of word-like objects which represent the smallest unit of linguistic meaning considered by a natural language processing system;
– *Lemmatisation* is the algorithmic process of determining the lemma for a given word. This phase substantially groups together the different inflected forms of a word so they can be analysed as a single item;
– *Stopwords elimination* phase filters out stop words from analysed text. Stop words usually refer to the most common words in a language, e.g. *the*, *is*, *at*, *which*, and so forth in english;
– *Part-of-speech tagging* attaches a tag denoting the part-of-speech to each word in a sentence, e.g., *Noun*, *Verb*, *Adverb*, etc.;
– *Named Entity Recognition* phase categorizes phrases (referred to as entities) found in text with respect to a potentially large number of semantic categories, such as person, organization, or geopolitical location;
– *Coreference Resolution* phase identify the linguistic expressions which make reference to the same entity or individual within a single document – or across a collection of documents.

The linguistic normalization also involves resolving multi-languages mismatching automatically or manually by translating metadata description from whatever languages into English.

3.3 Reference Model Matching

The main component of the framework is the Reference Model Matching function block. It is responsible for obtaining an alignment (A), i.e., a set of correspondences between the matched entities from the reference models (in this section and in the next one referred also as *input models*) and the *target model* (defined in Sect. 5). In this version of the framework, we mean as entities only ontology classes, so the framework does not apply to ontological individuals. This function block is also responsible for helping domain experts to select reference models from the Corpus by performing the *extended linguistic analysis* described later. Since the proposed approach is a multi-strategy based, the matcher involves three types of matching operations: string, linguistic and extended linguistic matching (see Fig. 1). It may use several sources as background knowledge, such as general background knowledge bases like WordNet and domain specific knowledge bases. All the matching subcomponents will be detailed in Sect. 4. Based on the similarity measure provided by the matching components for each pair of entities, the matcher implements a classification algorithm to predict how to relate entities each other. Here follows a brief description of each classification class and its meaning:

– *Equivalent* (in symbol: $\equiv$), the entities-pair is put in this class if they are supposed to be equivalent, i.e., they are supposed to have the same meaning;
– *Hypernym* (in symbol: $\supset$), the entities-pair is put in this class if the first term is supposed to be a broader concept w.r.t. the second one, i.e., it is an hypernym, or a superclass that subsumes the second;
– *Hyponym* (in symbol: $\subset$), the entities-pair is put in this class if the first term is supposed to be a narrower concept w.r.t. the second one, i.e., it is an hyponym, or a subclass of the second;
– *Related* (in symbol: $\cap$), the entities-pair is put in this class if the first term is supposed to be semantically related to the second one, or, thinking them set-theoretically, they have an intersection not null;
– *Disjointed* (in symbol: $\perp$), the entities-pair is put in this class if the first term is not related at all to the second one, or, thinking them set-theoretically, they have a null intersection;

3.4 Reference Models Merging or Integration

The mapping produced by the Aligner is used to integrate the selected input ontologies to the output ontology along with the target concepts. The concepts inside the target ontology can be envisioned as *hub* concepts, i.e., as a glue that holds together all the aligned concepts from the input ontologies. This approach is similar to that described in [36]. Figure 3 tries to explain it: each box represents a cluster of concepts equivalent to the hub concept and so equivalent to each other in turn. The red concept is the cluster representative and is used to create links with other clusters representative, this way creating the class-superclass hierarchy in the integration ontology. Introducing clustering is convenient to reduce computational load of the integrating algorithm because it avoid confronting each concept with each other concept of aligned ontologies but limit the confront to the cluster representative. In addition to clustering, another strategy that can be used to make scalable the integration algorithm in presence of very large ontologies is the *blocking* or *framing* technique. This consists in creating frames of clusters each relating to a knowledge domain or sub-domain, or to a specific aspect of the ontology. Thus, only the clusters in the same frame are confronted considerably reducing the effort for the integration phase. A similar approach can be also used for the matching and alignment stages of the framework.

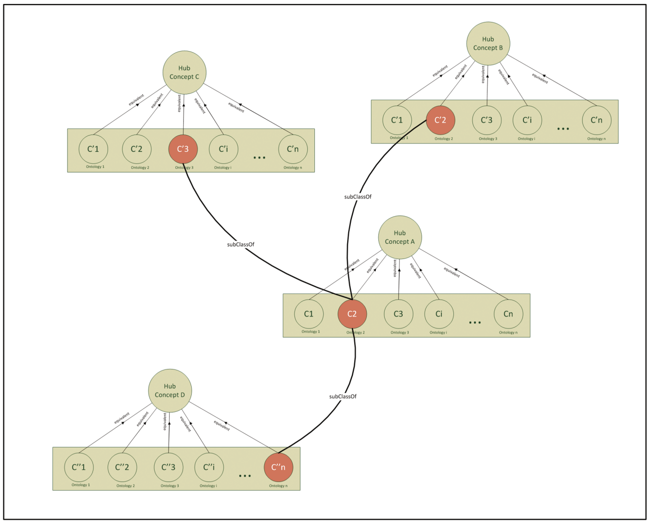


Fig. 3. The ontology integration strategy

An approach similar to that described in [10], instead, is used in this phase to merge or integrate input and target ontology entities: if two or more entities (concepts or relations) from the input models are equivalent w.r.t. a knowledge background ontology, from a set-theoretic perspective, they will be automatically merged in the same entity in the target ontology (m). If two entities are completely disjointed w.r.t all the knowledge background, they are considered irrelevant and so discarded d or can be added to the target in an ontology enrichment perspective (a). The domain experts consensus is also needed in the case in which an intersection exists between two entities w.r.t. the knowledge background (?). In the remaining cases the rules summarized in Table 1 will be used. In particular, when one entity subsumes ($\supseteq$) or is subsumed ($\subseteq$) by an entity in the knowledge background while the other is equivalent, the first will be integrated in the target ontology (i). Finally, when an entity is related somehow to an entity ($\cap$) in the knowledge background it will be discarded or added to target ontology with the consensus of the domain experts.

Table 1. Merging/integration operator rules

$\wedge$	$(=)$	$(\supseteq)$	$(\subseteq)$	$(\cap)$	$(\perp)$
$(=)$	m	i	i	?	d/a
$(\supseteq)$	i	?	?	?	d/a
$(\subseteq)$	i	?	i	?	d/a
$(\cap)$	?	?	?	?	d/a
$(\perp)$	d/a	d/a	d/a	d/a	d/a

4 The Matching Methodology

The objective of the proposed matching methodology consists in creating an alignment, i.e., a set of correspondences between the entities coming from the input models and those of the target ontology. In this work, the target ontology can be envisioned as the final goal of the integration approach but at the early phases of the proposed approach also as a kind of *proto-ontology* encompassing domain keywords, terms definitions, and concepts meanings related to the target knowledge domain and the application requirements upon which the domain experts agree. The alignment is used as a basis for the integration of the input ontologies into a coherent and global conceptualization of the domain under study.

We define the correspondence c as the tuple:

$$c = (e1, e2, r, v)$$

$e1$ being an entity from the first ontology, $e2$ being an entity from the second ontology to be compared, r being a matching relation and v being a similarity measure between the two entities based on the particular matching relation. Each correspondence can be expressed in different format, such as RDF/OWL, XML, or text. In this work, we use an RDF/OWL-based representation of alignments. Each entity involved in a matching operation has a unique id corresponding to the URI of the input ontology.

In the following sub-sections each of the matching methodologies will be further detailed.

4.1 The String-Based Matching

The string-based matching operation is performed between the labels attached to the entities (classes, relationships and properties) coming from the input and the target ontology. Additionally, metadata like comments, abstracts or descriptions will be taken into consideration in this phase. Concerning the string relatedness measures it is worth to note that the standard measurement, like the edit or Levenshtein distance works fine for very short strings (such as a single word), but this approach is far too sensitive to minor differences in word order, missing or extra words, and other such issues, so it is necessary to use *fuzzy* approaches and heuristics in order to relax the standard measures and avoid the risk of getting bad matchings. The fuzzy approach to string matching includes the following strategies:

- *Partial String Similarity*, we use the "best partial" heuristic when two strings are of noticeably different lengths. If the shorter string is of length m, and the longer string is length n, this similarity basically scores the best matching length-m sub-string. So, for example, the string "Yankees" and "New York Yankees" are a perfect partial match;

– *Out of Order* is another issue encountered with string matching. In this case two strings are similar if they differ only on the order of the terms within them. Two different approaches can be used here: *The token sort* approach involves tokenizing the string in question, sorting the tokens alphabetically, and then joining them back into a string, and *The token set* approach is similar, but a little bit more flexible. Here, we tokenize both strings, but instead of immediately sorting and comparing, we split the tokens into two groups: intersection and remainder. We use those sets to build up a comparison string.

Specifically, the fuzzy string similarity methods correct the standard measures by reducing their sensitiveness to minor differences in word order, missing or extra words, and other such issues. In this work we use a Levenshtein distance (i.e., the *edit distance*) to quantify how dissimilar two single-word labels are dissimilar to one another, and use others *fuzzy string matching* methods to compare multi-word labels or abstract and comments (with some tens of words at most).

4.2 The Linguistic Matching

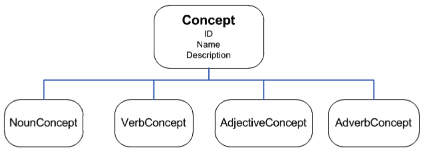
The linguistic matching is responsible for a comprehensive analysis of the terms used in the input models at a semantic level, using an external linguistic database like WordNet or other domain specific knowledge sources as background knowledge. In WordNet, nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a different concept [16]. The synsets are interlinked by conceptual semantic and lexical relations, thus realizing a graph-based structure where synsets are nodes and lexical-relations are edges. Exploiting the WordNet graph-based representation, it is possible to relate concepts at a semantic level, for example, by calculating the Wu-Palmer similarity [52], which counts the number of edges between two concepts by taking into account their proximity to the root concept of the hierarchy. According to [27], Wu-Palmer similarity has the advantage of being simple to calculate, in addition to its performances while remaining as expressive as the others.

Both the string and semantic similarity measure as a result of matching operation contribute to defining the semantic relation between the entities. The semantic relations correspond to set-theoretic relations between ontology classes: equivalence ($=$), disjointness ($\perp$), less general ($\leq$), more general ($\geq$) and concept correlation ($\cap$). A thresholding method is used to establish the type of set-theoretic relation that hold between the entities.

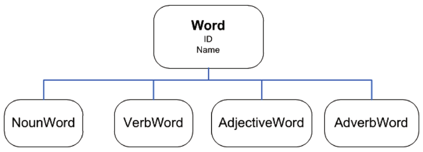
4.3 Extended Linguistic Matching

The extended linguistic matcher component defines and implements a meta-model for ontology matching using a conceptualization as much as possible close to the way in which the concepts are organized and expressed in human language [37]. The matcher exploits the meta-model for improving the accuracy for candidate reference model analysis. We define our meta-model as composed by a triple $\langle S; P; C \rangle$ where: S is a set of objects; P is the set of properties used to link the objects in S; C is a set of constraints on P. In this context, we consider concepts and words as objects; the properties are linguistic relations and the constraints are validity rules applied on linguistic properties w.r.t. the considered term category (i.e., noun, verb, adjective, adverb). In our approach, the target knowledge is represented by the target ontology. A concept is a set of words which represent an abstract idea. In this model, every node, both concept and word, is an OWL individual. The connecting edges in the ontology are represented as *ObjectProperties*. These properties have some constraints that depend on the syntactic category or on the kind of property (semantic or lexical). For example, the hyponymy property can relate only nouns to nouns or verbs to verbs; on the other hand a semantic property links concepts to concepts and a syntactic one relates word forms to word forms. Concept and word attributes are considered with *DatatypeProperties*, which relate individuals with a predefined data type. Each word is related to the represented concept by the ObjectProperty *hasConcept* while a concept is related to words that represent it using the ObjectProperty *hasWord*. These are the only properties able to relate words with concepts and vice versa; all the other properties relate words to words and concepts to concepts. Concepts, words and properties are arranged in a class hierarchy, resulting from the syntactic category for concepts and words and from the semantic or lexical type for the properties.

Figures 4(a) and (b) show that the two main classes are: Concept, in which all the objects have defined as individuals and Word which represents all the terms in the ontology.



(a) Concept

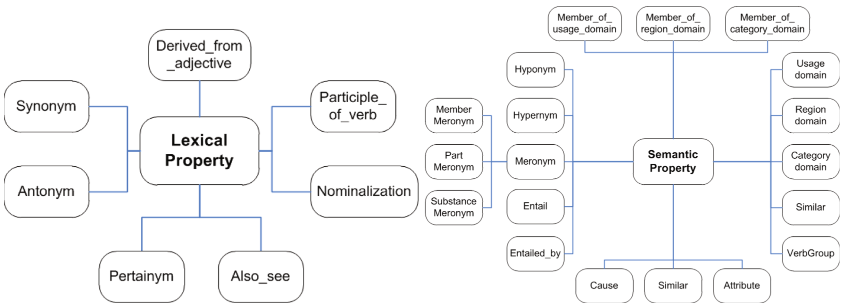


(b) Word

Fig. 4. Concept and Word

The subclasses have been derived from the related categories. There are some union classes useful to define properties domain and codomain. We define some attributes for Concept and Word respectively: Concept *hasName* that represents the concept name; *Description* that gives a short description of concept. On the other hand Word has Name as attribute that is the word name. All elements have an ID within the WordNet offset number or a user defined ID. The semantic and lexical properties are arranged in a hierarchy (see Fig. 5(a) and (b)). In Table 2 some of the considered properties and their domain and range of definition are shown.

The use of domain and codomain reduces the property range application. For example, the hyponymy property is defined on the sets of nouns and verbs;



(a) Lexical Properties (b) Semantic Properties

Fig. 5. Linguistic properties

Table 2. Properties

Property	Domain	Range
hasWord	Concept	Word
hasConcept	Word	Concept
hypernym	NounsAnd	NounsAnd
	VerbsConcept	VerbsConcept
holonym	NounConcept	NounConcept
entailment	VerbWord	VerbWord
similar	AdjectiveConcept	AdjectiveConcept

if it is applied on the set of nouns, it has the set of nouns as range, otherwise, if it is applied to the set of verbs it has the set of verbs as range. In Table 3 there are some of defined constraints and we specify on which classes they have been applied w.r.t. the considered properties; the table shows the matching range too.

Sometimes the existence of a property between two or more individuals entails the existence of other properties. For example, being the concept dog a hyponym of animal, we can assert that animal is a hypernymy of dog. We represent this characteristics in OWL, by means of property features shown in Table 4.

Table 3. Model constraints

Costraint	Class	Property	Constraint range
AllValuesFrom	NounConcept	hyponym	NounConcept
AllValuesFrom	AdjectiveConcept	attribute	NounConcept
AllValuesFrom	NounWord	synonym	NounWord
AllValuesFrom	AdverbWord	synonym	AdverbWord
AllValuesFrom	VerbWord	also_see	VerbWord

Table 4. Property features

Property	Features
hasWord	*inverse* of hasConcept
hasConcept	*inverse* of hasWord
hyponym	*inverse* of hypernym; *transitivity*
hypernym	*inverse* of hyponym; *transitivity*
cause	*transitivity*
verbGroup	*symmetry* and *transitivity*

Having defined the meta-model previously described, a Semantic Network (i.e., SN) is dynamically built using a dictionary based on WordNet or other domain specific resources. We define a semantic network as a graph consisting of nodes which represent concepts and edges which represent semantic relations between concepts. The role of domain experts is strategic in this phase because they interact with the system by providing a list of domain keywords and concept definition feeding the *proto-ontology*.

The SN is built starting from such first version of the target ontology, i.e., the domain keywords and the concept definition words sets. We then consider all the component synsets and construct a hierarchy, only based on the hyponymy property; the last level of our hierarchy corresponds to the last level of WordNets one. After this first step, we enrich our hierarchy considering all the other kinds of relationships in WordNet. In our approach, after the SN building step, we compare it with the selected input models lexical chains. The intersection between SN and the reference models gives us a lexical chain with the relevant terms related to the target ontology. All terms are linked by properties from the SN. Therefore, the SN give us a conceptual frame useful to discriminate the pertinent reference models from the other ones. In order to evaluate the relevancy of the selected reference model, it is necessary to define a system grading that is able to assign a vote to the model based on their syntactic and semantic content. We use the approach described in [37] to calculate a *Global Grade* (GG) for each semantic network related to each selected reference model. The GG is given by the sum of the *Syntactic-Semantic Grade* (SSG) and the *Semantic Grade* (SG). The first contribution gives us information about the analyzed model by taking into account the polysemy of the term, i.e., the measure of ambiguity in the use of a word, thus, with an accurate definition of the role of the considered term in the model. We call this measure *centrality* of the term i and we define it as: $\varpi(i) = \frac{1}{poly(i)}$ where *poly(i)* is the polysemy (number of senses) of i.

We can define the relevance of the reference model as the sum of its relevant word *weights* (terms centralities):

$$SSG(\nu) = \sum_{i=1}^{n} \varpi(i) \tag{1}$$

where n is the number of terms in the model ν.

The other contribution (SG) is based on a combination of the path length (l) between pairs of terms and the depth (d) of their subsumer (i.e., the first common ancestor), expressed as number of hops. Moreover, to each linguistic property, represented by arcs between the nodes of the SN, a weight is assigned in order to express the strength of each relation. We argue that not all the properties have the same strength when they link concepts or words (this difference is related to the nature of the considered linguistic property). The weights are real numbers in the [0,1] interval and their values are set by experiments and they are validated, from a strength comparison point of view, by experts.

We can now introduce the definition of *Semantic Grade* (SG), that extends a metric proposed in [26]:

$$SG(\nu) = \sum_{(w_i,w_j)} e^{-\alpha \cdot l(w_i,w_j)} \frac{e^{\beta \cdot d(w_i,w_j)} - e^{-\beta \cdot d(w_i,w_j)}}{e^{\beta \cdot d(w_i,w_j)} + e^{-\beta \cdot d(w_i,w_j)}} \qquad (2)$$

where (w_i, w_j) are a pairs of word in the intersection between DSN and model reference ν and $\alpha \geq 0$ and $\beta > 0$ are two scaling parameters whose values have been defined by experiments.

The final grade is the sum of the Syntactic-Semantic Grade and the Semantic Grade.

Once we have obtained the Global Grade for each semantic network, they are compared with a threshold value that act as a filter for the input reference models, thus giving us the most relevant input model at a linguistic level.

5 A Case Study from the Food Domain

In this section, we apply the proposed approach to the *food* domain, specifically to the *industrial production of food.* Since the food is an umbrella topic involving concepts related to different disciplines and applications, it represents a valid benchmark to test our approach in order to select those reference models that not only are about food in general, but whose main concern is on the production of food. The term "food" in fact can be found in reference models such as recipes for dishes served in a restaurant, biomedical thesaurus, commercial products catalogues and many others. Thus, the main result we expect here is to select the reference model for food that best fits the specific domain we are going to shape. Each of the function block in Fig. 1 will be applied in the follow. The collaboration with domain experts is precious in the first phase of the approach in order to individuate knowledge sources from which to extract models. Google search engine, Google Scholar, ISO International Classification of Standards and OAEI Iniziative Food test cases suit best in this case. Also specialized portal like BioPortal have been taken into consideration. The harvesting of reference models has been executed mostly manually even though some tools for automatizing search queries over Google Scholar have been successfully experimented[1].

[1] scholar.py, A parser for Google Scholar, written in Python. Available online: https://github.com/ckreibich/scholar.py.

As a result of applying this phase, many reference models, gathered in the reference *corpus*, have been collected. In order to select the best ones from the corpus the evaluation criteria discussed in Sect. 3 has been applied. In this regard, a greater weight has been given to reference models constructed in OWL or RDF language, these ones being the final languages used in the integrated ontology, and a greater weight has also been given to availability (open model data have been preferred) and model provenance (a high rank has been given to standard knowledge model like ISO standard). In Appendix A a brief description of each reference model is provided. Going forward, the reconciliation and normalization function block obtains a set of normalized, language-agnostic and de-structured knowledge models. In order to do this, we have firstly flattened the models concept hierarchy since we do not apply any structural analysis in this phase. Later on, we have applied the linguistic normalization operations listed in Sect. 3, to the model signature, i.e., to the textual representation of the model entities (concepts or relations) and their metadata (label, comments). This process, conveniently applied to each selected model, has resulted in a lexical chain for each model. Figure 6(a) shows the application of these steps to an excerpt of the National Cancer Institute Thesaurus (NCIT). In order to apply the matching function block to the input models, it is necessary to construct a prototype of the target ontology. In this work, the target ontology can be envisioned as the final goal of the integration approach but, also, at the early phases of the proposed methodology also as a kind of *proto-ontology*, i.e., a raw set of domain keywords, terms, definitions or in general concepts related to the knowledge domain under study without a formal structure. It is not an ontology in the strict sense of the term. It represents a set of grounded concepts related to the knowledge domain from where to start to aggregating concepts from the top ranked ontologies matched throughout the framework components. The contribution of domain expert users is important in producing the target ontology because the choice of terms or concepts involved in this phase will affect heavily the choice of the knowledge repositories and the input ontologies to be collected in the *reference model corpus*. The target ontology will provide the terms against which to match all the terms in the lexical chains coming from the input ontologies.

The third block helps knowledge engineers to automatically select and evaluate the input ontologies, on the basis of the target ontology, while providing a set of alignments, which will be used in the integration module. It performs string and linguistic matching between the lexical chains of the input models and that of the proto-ontology. Then it computes the Jaccard measure [15] for each model as the ratio between the cardinalities of the intersection of the input lexical chain and the target one and their union. The intersection of the input lexical chains contains all the terms having a string and a linguistic similarity measure greater than a prefixed threshold. According to the linguistic and string analysis, the best models related to the target ontology in this case study are: 1), 2) and 10). This result is consistent with the fact that the remaining ontologies or models listed in Table 5(a) are about others aspects of food mostly related

Table 5. Selected reference models analysis

(a) Qualitative criteria analysis

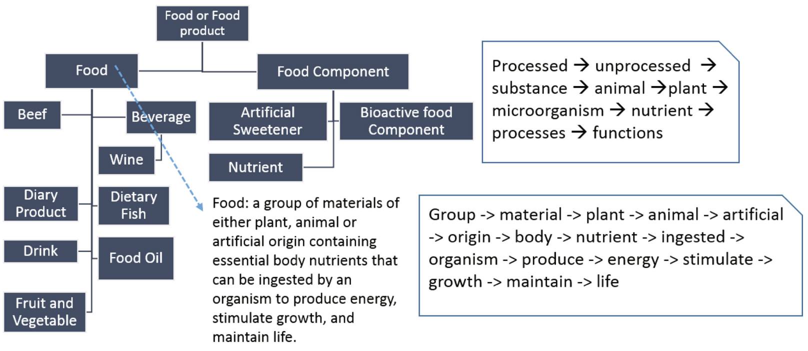
N.	Reference model	Formality	Generality	Structure	Language	Provenance	License
1)	National Cancer Institute Thesaurus	Formal	Domain	Ontology	OWL	Non-stand	Open
2)	AGROVOC	Semi-formal	Domain	Ontology	RDF	Non-stand.	Open
3)	Linked Recipe Schema	Semi-formal	Domain	Ontolog	RDF	Other	Open
4)	BBC Food Ontology	Semi-formal	Domain	Ontology	RDF	Other	Open
5)	LIRMM	Semi-formal	Domain	Ontology	RDF	Other	Open
6)	The Product Types Ontology	Semi-formal	Application	Ontology	RDF	Non-stand.	Open
7)	oregonstate.edu Food Glossary	Informal	Application	Glossary	Text	Other	Open
8)	Eurocode 2 Food Coding System	Informal	Domain	Classification	Text	Non-stand.	Open
9)	WAND Food and Beverage Taxonomy	Semi-formal	Domain	Taxonomy	Text	Private companies	Proprietary
10)	Food technology ISO Standard	Semi-formal	Domain	Taxonomy	Text	Stand. Organiz.	Proprietary

(b) Linguistic Matching Analysis

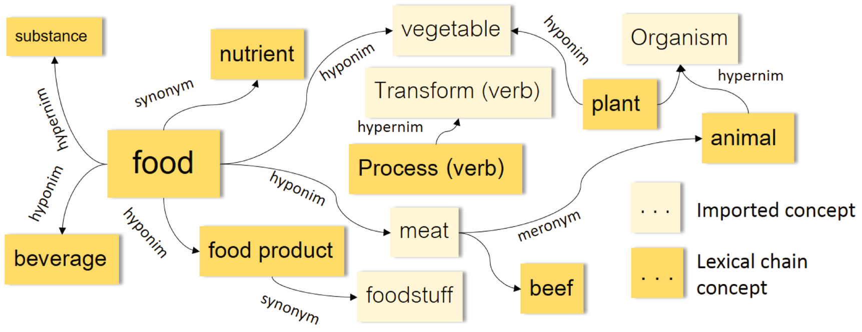
NCIT	Rel.	AGROVOC
Food Product	$\leq$	Food
Food Component	$=$	Food Composition
Beef	$\leq$	Animal Product
Beef	$\leq$	Fresh Meat
Beverage	$\geq$	Alcoholic Beverage
Wine	$=$	Wine
Drink	$\leq$	Alcoholic Beverage
Fruit and Vegetables	$\geq$	Vegetable Product
Nutrient	$\cap$	Food composition
Diary product	$\geq$	Animal Product

with food service or receipts (3), or about others not relevant aspects of the food domain (7, 8).

The final step of the matching function block is the extended linguistic analysis. As described in Sect. 4.3, this analysis converts the input and target lexical chains in semantic networks containing an extended set of concepts w.r.t. the initial set from the lexical chain. This new set encompasses hyponims, hypernims, meronims, holonims, etc., retrieved from WordNet. Furthermore, the concepts of the semantic network are linked to each other with the linguistic and semantic relations provided in the meta-model described in Sect. 4.2. Figure 6(b) shows an excerpt of the Extended Direct Semantic Network (DSN) for the NCIT lexical chain. The extended linguistic analysis has substantially confirmed the choice of the previously selected model giving more relevance to 10) and that is consistent with nature of ISO knowledge model. For this reason, the input models 1), 2) and 10) have been selected as the local ontologies to be integrated by the merging and integration function block. The fourth function block applies the alignments resulting from the matcher according to the Sect. 4.2 and integrate the local ontologies in the global one. Table 5(b) shows an excerpt of the alignment resulting by matching the NCIT and the AGROVC ontologies


Food or Food product
Food
Food Component
Beef
Beverage
Artificial Sweetener
Bioactive food Component
Wine
Nutrient
Diary Product
Dietary Fish
Drink
Food Oil
Fruit and Vegetable
Food: a group of materials of either plant, animal or artificial origin containing essential body nutrients that can be ingested by an organism to produce energy, stimulate growth, and maintain life.
Processed → unprocessed → substance → animal →plant → microorganism → nutrient → processes → functions
Group -> material -> plant -> animal -> artificial -> origin -> body -> nutrient -> ingested -> organism -> produce -> energy -> stimulate -> growth -> maintain -> life

(a) Ontology Excerpt and Lexical Chain



(b) Extended Semantic Network Excerpt

Fig. 6. NCIT Extended Linguistic analysis

6 Experimental Results

Since the methodology described in this paper applies to different phases of knowledge reuse and integration, and every phase produces intermediate results, which would require specific comparisons with other similar matching systems or approaches, our discussion here concentrates on a global level, i.e., how efficiently and effectively, in terms of *precision* and *recall*, the candidate reference models have been selected to be reused in the integration phase of the workflow, considering the methodology as a whole. We leave to further investigations and measurements a detailed comparison of results coming from the matching modules and the integration module.

For the sake of our global analysis, we use the standard definitions of precision and recall by introducing two sets of reference models, namely, the relevant reference models set (described later) and the retrieved reference models set, i.e., those resulting by applying the proposed methodology. The relevant models have been individuated manually, by averaging the score assigned them by a group of experts. This task has led us to select model 1), 2), 6) and 10) as relevant.

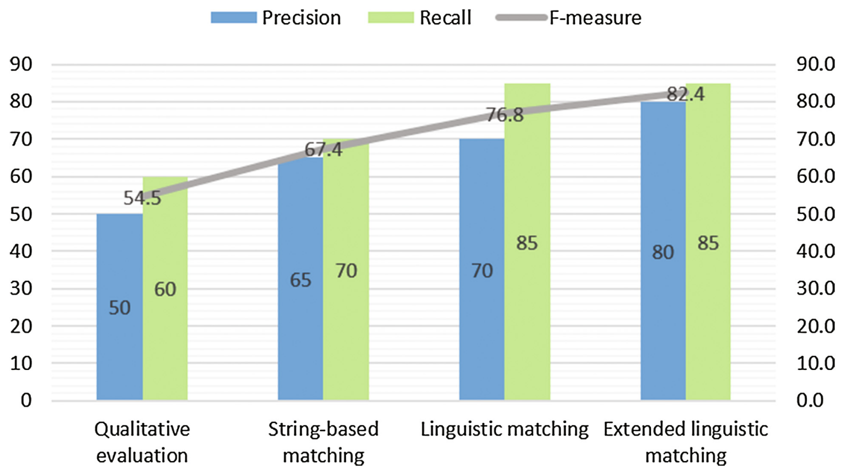


Fig. 7. Reference models selection results

They represent a kind of ground truth for the analysis described here. Figure 7 shows an overview of the experimental results by plotting the maximum values of precision, recall and F-measure in four different moments of the whole procedure, namely, at the end of the first phase (reference models retrieval) after applying the qualitative criteria to make a first selection of relevant models, at the end of string-based matching phase, at the end of the linguistic-based matching phase and, finally, at the end of the extended linguistic-based matching phase. The figure shows a common behaviour regarding the precision and recall curves: while the first measure increases, the second one decreases. In this context, this is due to the effect of applying increasingly sophisticated matching methodologies in order to retrieve the candidate reference models, that ameliorate the precision at the expense of the recall. The F-measure continuously increases throughout the matching phases as result of the precision rise, despite of the fluctuating trend of the recall.

Given that a detailed comparison of the alignments outcoming from the matching modules will be subject of future investigations, as discussed in the conclusions section, a brief outline of the methodology under study is provided as follows. Starting from a common approach in the literature, sets of reference documents are replaced by sets of correspondences, i.e., alignments. The alignment (A) returned by the matching module to evaluate is compared to a reference alignment (R). This latter, meant like a ground truth, is obtained manually with the involvement of domain experts a can include a relevant subset of the entire alignment between two ontologies. Like in information retrieval, precision measures the ratio of correctly found correspondences (true positives) over the total number of returned correspondences (true positives and false positives). In logical terms, this is supposed to measure the correctness of the method. Precision and recall together with the F-measure are commonplace measures in information retrieval and they have also been adapted for ontology alignment evaluation [11] but they have the drawback that whatever correspondence has not been found is definitely not considered. As a result, they do not discriminate between a bad and a better alignment. So, following the approach in [12], instead of comparing alignments set-theoretically, we will measure the proximity of correspondence sets rather than the strict size of their overlap, in other words, instead of taking the cardinal of the intersection of the two sets ($| \text{R} \cap \text{A}|$), the natural generalizations of precision and recall measure their proximity ω(A,R), where ω is an overlap function between alignments based on a proximity function (σ) between two correspondences, as defined in the relaxed Precision and Recall measures discussed in [12]. Furthermore, since our goal is to design a generalization of precision and recall that is not only proximity-based but also semantically grounded, we will consider as set of alignments the initial one plus the correspondences that are *consequences* of the evaluated alignments (as recalled) and those that are consequence of the reference alignments (R) (as correct), being the notion of *consequences* defined in [14]. The characterization of the ω and the σ function is out of the scope of this work and will be subject to further investigations in future works.

7 Conclusions

A multi-strategy approach, manual and automatic, in ontology reuse and integration may result in a feasible and useful practice. The experimentation within the Food domain has demonstrated that an approach based on linguistic matching can help to automatize the selection of the most relevant reference models, by properly distinguishing those models that belong to a specific interpretation of the domain under study among others, and the integration of the local ontologies in the global model. Nonetheless, this process requires a significant amount of manual work, even when it deals with common and formal models. This requirement may be a severe limitation for a widespread adoption of the knowledge reuse and may represent a relevant technological gap to be addressed by researchers in the near future. Furthermore, it is worth to test the proposed approach against other knowledge domains in order to evaluate its practicability in different contexts. New matching similarity measures will be subject of further researches with the aim of improving the precision of the alignments, comparing them with gold standard tests, and this way ameliorating the entire approach. In this regard a methodology for evaluating alignment and matching algorithms similar to that proposed by the Ontology Alignment Evaluation Initiative [13] or others like [23] can be adopted. Furthermore, a synergistic use of information visualization techniques and the capabilities of new tools emerged in the landscape of Big Graph Data like Neo4J and its declarative graph query language (Cypher) [2,5], can help in visualize the alignment set and perform over it different kind of evaluation measurements, which adopt extended versions beyond the classical precision and recall measures, by exploiting its features like the pattern-based queries and its iconicity. Finally, most of the techniques used here can be adopted in knowledge-based systems also taking into consideration multimedia features [30,41,42].

A Appendix

The list below provides a short description for each selected reference model filtered out from the corpus of retrieved references.

1. **National Cancer Institute Thesaurus**[2] by the American National Institutes of Health (NIH):
   The NCI Thesaurus is a reference terminology and biomedical ontology used in NCI systems. It covers vocabulary for clinical care, translational and basic research, and public information and administrative activities. It contains 118941 classes, 46839 individuals, 173 properties, 16 as the max depth, 3235 children, an average number of children equal to 6 and 36013 classes with no definitions
2. **AGROVOC Multilingual agricultural thesaurus**[3] by AIMS Advisory Board:
   AGROVOC is a controlled vocabulary covering all areas of interest of the Food and Agriculture Organization (FAO) of the United Nations, including food, nutrition, agriculture, fisheries, forestry, environment etc. AGROVOC consists of over 32,000 concepts available in 27 languages: Arabic, Burmese, Chinese, Czech, English, French, German, Hindi, Hungarian, Italian, Japanese, Khmer, Korean, Lao, Malay, Moldovian, Persian, Polish, Portuguese, Russian, Slovak, Spanish, Telugu, Thai, Turkish, Ukrainian, Vientamese.
3. **Linked Recipe Schema**[4] by schema.org:
   Schema.org is a collaborative, community activity with a mission to create, maintain, and promote schemas for structured data on the Internet, on web pages, in email messages, and beyond. These vocabularies cover entities, relationships between entities and actions, and can easily be extended through a well-documented extension model.
4. **BBC Food Ontology**[5] by BBC:
   The Food Ontology is a simple lightweight ontology for publishing data about recipes, including the foods they are made from and the foods they create as well as the diets, menus, seasons, courses and occasions they may be suitable for. Whilst it originates in a specific BBC use case, the Food Ontology should

[2] National Cancer Institute Thesaurus. Available online, https://ncit.nci.nih.gov/ncitbrowser/.

[3] AGROVOC Multilingual agricultural thesaurus. Available online, http://aims.fao.org/vest-registry/vocabularies/agrovoc-multilingual-agricultural-thesaurus.

[4] Linked Recipe Schema. Available online, http://aims.fao.org/vest-registry/vocabularies/agrovoc-multilingual-agricultural-thesaurus.

[5] BBC Food Ontology. Available online, http://www.bbc.co.uk/ontologies/fo.

be applicable to a wide range of recipe data publishing across the web. It presents 57 named classes.

5. **LIRMM Food Ontology**[6] by LIRMM Laboratoire:
This ontology models the Food domain. It allows to describe ingredients and food products. Some classes are: food:Recipe, food:Food, food:FoodProduct, food:Dish, food:Ingredient, etc.
6. **The Product Types Ontology**[7] by E-Business and Web Science Research Group at Bundeswehr University Munich:
This ontology contains 300,000 precise definitions for types of product or services that extend the schema.org and GoodRelations standards for e-commerce markup.
7. **Oregon State Food Glossary**[8] by Oregon State University:
FoodON is a new ontology built to interoperate with the OBO Library and to represent entities which bear a "food role". It encompasses materials in natural ecosystems and food webs as well as human-centric categorization and handling of food.
8. **Eurocode 2 Food Coding System**[9] by European FLAIR Eurofoods-Enfant Project:
The Eurocode 2 Food Coding System was originally developed within the European FLAIR Eurofoods-Enfant Project to serve as a standard instrument for nutritional surveys in Europe and to serve the need for food intake comparisons. It contains 162 named classes.
9. **WAND Food and Beverage Taxonomy**[10] by WAND Company:
The WAND Food and Beverage Taxonomy includes 1,278 terms including foods, beverages, ingredients, and additives. This taxonomy includes anything that somebody may consume as food, including some prepared foods. The WAND Foods and Beverages Taxonomy is ideal for restaurants, groceries, and food manufacturers.
10. **Food technology ISO Standard**[11] by ISO:
International standard by ISO which provides a terminology for processes in the food industry, including food hygiene and food safety, food products in general, methods of tests and analysis for food ICS (International Classification for Standards) products, materials and articles in contact with foodstuffs and materials and articles in contact with drinking water, plants and equipment for the food industry.

[6] LIRMM Food Ontology. Available online, http://data.lirmm.fr/ontologies/food.

[7] The Product Types Ontology. Available online, http://www.productontology.org/.

[8] Oregon State Food Glossary. Available online, http://icbo.cgrb.oregonstate.edu/node/282.

[9] Eurocode 2 Food Coding System. Available online, http://www.danfood.info/eurocode/.

[10] WAND Food and Beverage Taxonomy. Available online, http://www.wandinc.com/wand-food-and-beverage-taxonomy.aspx.

[11] International classification for Standards (ISC). Available online, http://www.iso.org/iso/home/store/catalogue_ics.htm.

References

1. Bontas, E.P., Mochol, M., Tolksdorf, R.: Case studies on ontology reuse. In: Proceedings of the IKNOW 2005 International Conference on Knowledge Management, vol. 74 (2005)
2. Caldarola, E.G., Picariello, A., Rinaldi, A.M.: Big graph-based data visualization experiences: the wordnet case study. In: 2015 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K), vol. 1, pp. 104–115, November 2015
3. Caldarola, E.G., Picariello, A., Rinaldi, A.M.: An approach to ontology integration for ontology reuse in knowledge based digital ecosystems. In: Proceedings of the 7th International Conference on Management of computational and collective intElligence in Digital EcoSystems, pp. 1–8. ACM (2015)
4. Caldarola, E.G., Rinaldi, A.M.: An approach to ontology integration for ontology reuse. In: 2016 IEEE 17th International Conference on Information Reuse and Integration (IRI), pp. 384–393. IEEE (2016)
5. Caldarola, E.G., Rinaldi, A.M.: Improving the visualization of wordnet large lexical database through semantic tag clouds. In: 2016 IEEE International Congress on Big Data (BigData Congress), pp. 34–41. IEEE (2016)
6. Caldarola, E.G., Rinaldi, A.M.: Big data: a survey - the new paradigms, methodologies and tools. In: Proceedings of 4th International Conference on Data Management Technologies and Applications, pp. 362–370 (2015)
7. Cataldo, A., Rinaldi, A.M.: An ontological approach to represent knowledge in territorial planning science. Comput. Environ. Urban Syst. **34**(2), 117–132 (2010)
8. Chalupsky, H.: Ontomorph: a translation system for symbolic knowledge. In: KR, pp. 471–482 (2000)
9. Choi, N., Song, I.Y., Han, H.: A survey on ontology mapping. ACM SIGMOD Rec. **35**(3), 34–41 (2006)
10. Cruz, I.F., Xiao, H.: The role of ontologies in data integration. Eng. Intell. Syst. Electr. Eng. Commun. **13**(4), 245 (2005)
11. Do, H.H., Melnik, S., Rahm, E.: Comparison of schema matching evaluations. In: Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, pp. 221–237. Springer, Heidelberg (2002)
12. Ehrig, M., Euzenat, J.: Relaxed precision and recall for ontology matching. In: Proceedings of K-Cap 2005 Workshop on Integrating Ontology, pp. 25–32 (2005)
13. Euzenat, J., Ehrig, M., Castro, R.: Towards a methodology for evaluating alignment and matching algorithms. Technical report, Ontology Alignment Evaluation Initiative (OAEI) (2005)
14. Euzenat, J.: Semantic precision and recall for ontology alignment evaluation. In: IJCAI, pp. 348–353 (2007)
15. Euzenat, J., Shvaiko, P., et al.: Ontology Matching, vol. 18. Springer, Heidelberg (2007)
16. Fellbaum, C.: Wordnet. The Encyclopedia of Applied Linguistics (1998)
17. Flouris, G., Plexousakis, D., Antoniou, G.: A classification of ontology change. In: SWAP (2006)
18. Gaeta, M., Orciuoli, F., Paolozzi, S., Salerno, S.: Ontology extraction for knowledge reuse: the e-learning perspective. IEEE Trans. Syst. Man Cybern. Part A Syst. Hum. **41**(4), 798–809 (2011)

19. Ghazvinian, A., Noy, N.F., Musen, M.A.: How orthogonal are the obo foundry ontologies? J. Biomed. Semant. **2**(2), 1 (2011)
20. Giunchiglia, F., Shvaiko, P., Yatskevich, M.: S-match: an algorithm and an implementation of semantic matching. In: The Semantic Web: Research and Applications, pp. 61–75. Springer (2004)
21. Gruber, T.R.: A translation approach to portable ontology specifications. Knowl. Acquisition **5**(2), 199–220 (1993)
22. Heflin, J., Hendler, J.: Dynamic ontologies on the web. In: AAAI/IAAI, pp. 443–449 (2000)
23. Jean-Mary, Y., Kabuka, M.: Asmov: ontology alignment with semantic validation. In: Joint SWDB-ODBIS Workshop, pp. 15–20 (2007)
24. Kalfoglou, Y., Schorlemmer, M.: Ontology mapping: the state of the art. Knowl. Eng. Rev. **18**(1), 1–31 (2003)
25. Leung, N.K.Y., Lau, S.K., Fan, J., Tsang, N.: An integration-oriented ontology development methodology to reuse existing ontologies in an ontology development process. In: Proceedings of the 13th International Conference on Information Integration and Web-Based Applications and Services, iiWAS 2011, pp. 174–181. ACM, New York (2011). http://doi.acm.org/10.1145/2095536.2095567
26. Li, Y., Bandar, Z.A., McLean, D.: An approach for measuring semantic similarity between words using multiple information sources. IEEE Trans. Knowl. Data Eng. **15**(4), 871–882 (2003)
27. Lin, D.: An information-theoretic definition of similarity. In: ICML, vol. 98, pp. 296–304. Citeseer (1998)
28. McBride, B.: Jena: a semantic web toolkit. IEEE Internet Comput. **6**(6), 55 (2002)
29. McGuinness, D.L., Fikes, R., Rice, J., Wilder, S.: The chimaera ontology environment. In: AAAI/IAAI 2000, pp. 1123–1124 (2000)
30. Moscato, V., Picariello, A., Rinaldi, A.M.: A recommendation strategy based on user behavior in digital ecosystems. In: Proceedings of the International Conference on Management of Emergent Digital EcoSystems, pp. 25–32. ACM (2010)
31. Nathalie, A.: Schema matching based on attribute values and background ontology. In: 12th AGILE International Conference on Geographic Information Science, vol. 1(1), pp. 1–9 (2009)
32. Noy, N.F., Musen, M.A.: Anchor-prompt: using non-local context for semantic matching. In: Proceedings of the Workshop on Ontologies and Information Sharing at the International Joint Conference on Artificial Intelligence (IJCAI), pp. 63–70 (2001)
33. Noy, N.F., Musen, M.A.: Smart: Automated support for ontology merging and alignment. In: Proceedings of the 12th Workshop on Knowledge Acquisition, Modelling, and Management (KAW 1999), Banf, Canada (1999)
34. Noy, N.F., Musen, M.A.: Algorithm and tool for automated ontology merging and alignment. In: Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000). Available as SMI technical report SMI-2000-0831 (2000)
35. Pinto, H.S., Martins, J.P.: Ontologies: how can they be built? Knowl. Inf. Syst. **6**(4), 441–464 (2004)
36. Rahm, E.: The case for holistic data integration. In: East European Conference on Advances in Databases and Information Systems, pp. 11–27. Springer, Cham (2016)
37. Rinaldi, A.M.: A content-based approach for document representation and retrieval. In: Proceedings of the Eighth ACM Symposium on Document Engineering, pp. 106–109. ACM (2008)

38. Rinaldi, A.M.: An ontology-driven approach for semantic information retrieval on the web. ACM Trans. Internet Technol. (TOIT) **9**(3), 10 (2009)
39. Rinaldi, A.M.: Improving tag clouds with ontologies and semantics. In: 2012 23rd International Workshop on Database and Expert Systems Applications, pp. 139–143. IEEE (2012)
40. Rinaldi, A.M.: Document summarization using semantic clouds. In: 2013 IEEE Seventh International Conference on Semantic Computing (ICSC), pp. 100–103. IEEE (2013)
41. Rinaldi, A.M.: A multimedia ontology model based on linguistic properties and audio-visual features. Inf. Sci. **277**, 234–246 (2014)
42. Rinaldi, A.M.: A complete framework to manage multimedia ontologies in digital ecosystems. Int. J. Bus. Process Integr. Manag. **7**(4), 274–288 (2015)
43. Schreiber, G.: Knowledge Engineering and Management: The CommonKADS Methodology. MIT Press, Cambridge (2000)
44. Shah, T., Rabhi, F., Ray, P., Taylor, K.: A guiding framework for ontology reuse in the biomedical domain. In: 2014 47th Hawaii International Conference on System Sciences, pp. 2878–2887. IEEE (2014)
45. Shamdasani, J., Hauer, T., Bloodsworth, P., Branson, A., Odeh, M., McClatchey, R.: Semantic matching using the UMLS. In: The Semantic Web: Research and Applications, pp. 203–217. Springer, Heidelberg (2009)
46. Shvaiko, P., Euzenat, J.: Ontology matching: state of the art and future challenges. IEEE Trans. Knowl. Data Eng. **25**(1), 158–176 (2013)
47. Smith, B., Ashburner, M., Rosse, C., Bard, J., Bug, W., Ceusters, W., Goldberg, L.J., Eilbeck, K., Ireland, A., Mungall, C.J., et al.: The obo foundry: coordinated evolution of ontologies to support biomedical data integration. Nat. Biotechnol. **25**(11), 1251–1255 (2007)
48. Stumme, G., Maedche, A.: FCA-merge: Bottom-up merging of ontologies. In: IJCAI, vol. 1, pp. 225–230 (2001)
49. Suárez-Figueroa, M.C., Gómez-Pérez, A., Motta, E., Gangemi, A.: Ontology Engineering in a Networked World. Springer Science & Business Media, Heidelberg (2012)
50. (W3C), W.W.W.C.: W3c semantic web activity. Technical report (2011)
51. Wiederhold, G.: Large-scale information systems. In: Database Applications Semantics, p. 34 (2016)
52. Wu, Z., Palmer, M.: Verbs semantics and lexical selection. In: Proceedings of the 32nd Annual Meeting on Association for Computational Linguistics, pp. 133–138. Association for Computational Linguistics (1994)
53. Xiang, Z., Courtot, M., Brinkman, R.R., Ruttenberg, A., He, Y.: Ontofox: web-based support for ontology reuse. BMC Res. Notes **3**(1), 1 (2010)
54. Zedlitz, J., Luttenberger, N.: Transforming between UML conceptual models and owl 2 ontologies. In: Terra Cognita 2012 Workshop, vol. 6, p. 15 (2012)

Classifier Fusion by Judgers on Spark Clusters for Multimedia Big Data Classification

Yilin Yan[1], Qiusha Zhu[2], Mei-Ling Shyu[1](✉), and Shu-Ching Chen[3]

[1] Department of Electrical and Computer Engineering,
University of Miami, Coral Gables, FL 33146, USA
y.yan4@umiami.edu, shyu@miami.edu
[2] Citibank, 1000 North West Street, Wilmington, DE 19801, USA
q.zhu2@umiami.edu
[3] School of Computing and Information Sciences,
Florida International University, Miami, FL 33199, USA
chens@cs.fiu.edu

Abstract. The exponential growth of multimedia data including images and videos has been witnessed on social media websites like Instagram and YouTube. With the rapid growth of multimedia data size, efficient processing of these big data becomes more and more important. Meanwhile, lots of classifiers have been proposed for a number of data types. However, how to assemble these classifiers efficiently remains a challenging research issue. In this paper, a novel scalable framework is proposed for classifier ensemble using a set of judgers generated based on the training and validation results. These judgers are ranked and put together as a hierarchically structured decision model. The proposed ensemble framework is deployed on an Apache Spark cluster for efficient data processing. Our experimental results on multimedia datasets containing different actions show that our ensemble work performs better than several state-of-the-art model fusion approaches.

Keywords: Classifier ensemble · Classifier fusion · Apache spark · Big data

1 Introduction

Content-based multimedia data retrieval and management have become a very important research area due to its broad applications in this century [1–5]. For instance, video content analysis, in the context of automatically analyzing human actions in videos, has been widely utilized in video event mining, video summarization, camera surveillance, etc. [6–11]. Meanwhile, the deluge of multimedia big data has made data-oriented research more and more important, especially in this big data era [12–17]. In this paper, we propose a data mining based framework to solve the problem of multimedia big data classification.

A number of data analysis technologies have been developed in the past decade, including a variety of classification algorithms for different kinds of datasets. Nevertheless, a single classifier can hardly handle heterogeneous media types from different datasets in various situations. Commonly speaking, an ensemble of data classifiers will

© Springer International Publishing AG 2018
S.H. Rubin and T. Bouabana-Tebibel (eds.), *Quality Software Through Reuse and Integration*, Advances in Intelligent Systems and Computing 561,
DOI 10.1007/978-3-319-56157-8_5

be always better than the individual ones as "Vox Populi, Vox Dei". To take advantages of different classifiers and reach the best performance on a dataset, lots of research groups recently focus on assembling useful classifiers together.

A novel idea of classification combination is proposed in this paper. Based on the confusion matrices of different classifiers, a scalable classifier ensemble framework assisted by several "judgers" is proposed to integrate the outputs from multiple classifiers for multimedia big data classification. These judgers are put together as a boosted classifier. Specifically, a set of "judgers" are generated based on training and validation results from different classifiers and features at first. On the second step, these judgers are ranked and organized into a hierarchically structured decision model. Finally, an Apache Spark-based classification system is developed which can be applied for multimedia big data classification.

In the testing stage, an instance is fed to different classifiers, and then the classification results are passed to the proposed hierarchical structured decision model to derive the final result. By running on a Spark cluster, the system is well designed for multimedia big data processing. Our experimental results on two popular video datasets including different actions show that the proposed work successfully fuses classifiers and outperforms several existing classifier ensemble frameworks.

Our proposed system has the following three main contributions.

- Novel concepts of positive and negative "judgers" are defined to assemble a novel hierarchical structured decision framework.
- The proposed work considers the fusion of classifiers using both the same features and different feature spaces simultaneously.
- A unified ensemble fusion model in a big data infrastructure using Spark is developed and uses action classification in videos as a proof-of-concept.
- Experimental results show promising results by comparing to several state-of-the-art methods.

This paper is organized as follows. Section 2 presents related work in classifier ensemble. In Sect. 3, the proposed classifier fusion model is introduced in details. In Sect. 4, two benchmark action datasets are used for evaluation. Finally, concluding remarks are presented in Sect. 5.

2 Related Work

2.1 Multiple Classifiers and Features

In the scope of multimedia data classification, multi-classifier fusion is an important research area because one classifier is unable to perform better than other classifiers on all types of data. In [18], the authors develop gradient histograms using orientation tensors for human action. A classifiers fusion based framework using statistical fusion such as GMM (Gaussian Mixture Model) fusion and ANN (Artificial Neural Network) fusion is proposed in [19]. While conflict results can be generated by different classifiers, previous results have indicated that the fusion of multiple different results can improve the performance of individual classifiers. In general, classifiers ensemble is a good resolution to conflict classification results. However, how to find a good way to fuse these classification results from different classifiers remains as a big issue.

Meanwhile, how to utilize multiple features [20–26] by different feature extraction models from multimedia datasets is another hot research area in the past decade. Different classification models can be employed for different kinds of features, which may discover different properties of the data [27, 28]. The authors in [29] found the complementary nature of the descriptors from different viewpoints, such as semantics, temporal and spatial resolution. They also employed a hierarchical fusion that combines static descriptors and dynamic descriptors. Since high-level semantics are sometimes difficult to be captured by visual features, textual features were used in [30] and a sparse linear fusion scheme was proposed in their work to combine visual and textual features for semantic concept retrieval and data classification.

2.2 Multi-classifier Fusion

Generally speaking, the existing work on multi-classifiers ensembles models falls into four types as follows.

2.2.1 Weighted Combination Strategy

The weighted combination strategy is a popular and straight-forward strategy and commonly used in many classifiers ensemble models. Two examples are sum and product approaches as weighted combination rules. The sum rule treats the sum value as the arithmetic mean while the product rule treats the product value as the geometric mean. The sum rule is equivalent to the product rule for small deviations in the classifier outcomes under the same assumption [31]. In general, the product rule is good when the individual classifiers are independent.

Furthermore, the sum and product rules can be generalized to the weighted combination approaches for different scores. The key to this strategy is to find a suitable set of weights for different scores generated by different classifiers. Several different strategies were proposed in the past in order to determine the weights. For instance, an information gain method for assigning weights is explained in [32] where the authors adapt and evaluate existing combining methods for the traffic anomaly detection problem and showed that the accuracies of these detectors can be improved. Meng et al. [33] utilize the normalized accuracy to compute the weights for each model built on a specific image patch. In a recent study proposed in [34], the researchers further extend this method by first sorting all the models according to the interpolated average precision and then selecting the models with top performance. The number of models to retain in the final list is determined via an empirical study.

Although several experimental results indicate that sometimes weighted combination strategy can give a relatively good performance, the success of this kind of approaches relies on specific knowledge from domain experts or experience from data mining researchers to provide a good estimation of weights. This clearly shows the importance of proper choice of weights.

2.2.2 Statistics-Based Strategy

The sum rule can be also considered as a special statistics-based strategy. Some other commonly used approaches in statistics-based approaches are “sum”, “max”, “min”, and “median” rules. The “sum” strategy here is somewhat different from the sun rule in weight combination strategy and gives an estimation of the final score based on the majority-voting theory. By setting all the weights to the same value, it is then equivalent to the sum rule. The “max” fusion approach is a relatively conservative estimation, where the highest score of all the models is chosen. On the other hand, the “min” fusion strategy picks the lowest value. The fourth one, “median” rule, gets the median value of all the scores.

Kittler et al. [35] develop a common framework for classifier combination and show that many existing schemes can be considered as special cases of compound classification where all the features are used jointly to make a decision. In [36], Kuncheva evaluates the advantages and disadvantages of these strategies in details from a theoretical point of view. The main advantage of the statistics-based approach is the low time complexity, while the main disadvantage is that the performance of these models is not quite stable under the condition that the underlying models are not accurate.

2.2.3 Regression-Based Strategy

The regression-based strategy receives a lot of attentions recently. In this research direction, the logistic regression based model is commonly utilized. Parameters are estimated using the gradient descent approach in the training stage. After the parameters are learned, the score of a testing data instance can be computed. In [37], a novel logistic regression model is trained to integrate the scores from testing data by different classification models to get a final probabilistic score for the target concept.

Although the logistic regression model sometimes gives relatively robust performances in practice, the disadvantage of regression-based strategy is that this algorithm may suffer from the problem of overfitting.

2.2.4 Bayesian Probabilistic Strategy

With the assumption of the scores are conditionally independent with each other, the Bayesian theory is also widely used in multi-classifier fusion, and sometimes is combined with other strategies [38]. The most issue of this strategy is that the previous assumption does not hold under most circumstances. The final score is computed using the Bayesian rule based on all the scores from the models.

The theory of Dempster–Shafer is an improved method of the Bayesian theory for a subjective probability. It is also a powerful method for combining measures of evidence from different classifiers. The authors in [39] develop another classifier combination technique based on the Dempster–Shafer theory of evidence by adjusting the evidence of different classifiers based on minimizing the MSE of training data. However, this kind of approaches may still give relatively bad performance because of the severe deviation from the independence assumption in real cases.

3 The Proposed Framework

3.1 Feature Extraction

Different from some other papers that extract features from the whole images or frames, we do feature extraction only from the Region of Action (ROA) in order to capture the action related information. Here, we use the ROA selection and feature extraction strategy from [21] which improve the action detection and recognition performance in an automated system by fully exploring the ROAs. This cited work also analyzes and integrates the motion information of actions in both temporal and spatial domains.

This approach can be roughly divided into three steps. In the first step, ROAs are driven from two popular spatio-temporal methods including Harris3D corners and optical flow. Next, the idea of integral image in [40] is utilized for its fast implementation of the box type convolution filters. Similar idea is also used in SURF [41] promoted from SIFT [42]. Finally, the Gaussian Mixture Models (GMM) are applied sequentially in this paper. All the mean vectors of the Gaussian components in the generated GMM model are concatenated to create GMM supervectors for video action recognition. SIFT and STIP [43] features which are widely used are extracted from frames in video datasets in our work to describe the action sequences in video action recognition. Other good features can be also fed to the proposed model for better results.

3.2 Classification

Similar to features, our classifiers fusion framework accepts most kinds of classifiers. Specifically, three popular classification algorithms are used in this paper as follows:

3.2.1 Support Vector Machine (SVM)

SVM is one of the state-of-the-art algorithms for classification in the data mining area. The general idea is to build a separating hyperplane to classify the data instances so that the geometric margin is maximized. In order to handle the case that the classes are linearly inseparable. The kernel trick is utilized. In this paper, we applied the LibSVM, which is one of the most popular off-the-shelf software implementations. The radial basis function (RBF) kernel is chosen based on experimental results of an empirical study.

3.2.2 Sparse Representation

Sparse representation [44] is a hot research topic in the past decade which builds overcomplete dictionaries to represent the training dataset. With this kind of dictionaries including prototype signal-atoms, signals can be described by sparse linear combinations of these atoms. Sparse representation has been widely used in the areas of image denoising, object detection, semantic concept retrieval, information compression and other useful applications including multimedia data classification.

Here, we use Sparse Representation Classification (SRC) along with its dictionary learning techniques and design a framework to analyze actions of one person and events between multiple people. For the task of dictionary learning, the widely-used

K-SVD algorithm is adopted, which aims at deriving the dictionary of sparse representation using the Singular Value Decomposition (SVD). The class label of a testing instance can be determined by finding the minimum reconstruction error of the testing sample represented by the trained dictionaries. A disadvantage of the SRC scheme is its high computational complexity associated with the minimization problem.

3.2.3 Hamming Distance

A novel scheme Hamming Distance Classification (HDC) [45] is also included in our work. HDC is an efficient classifier for real-time applications due to its efficiency. For each class, a threshold will be calculated by the median value of the inner products of each pair of features in the training set. Given a testing instance, a binary string can be coded based on the inner product between the testing sample and each training sample. If the inner product of the testing sample and a training sample is less than the trained threshold, it would be assigned to "0"; otherwise, it would be coded as "1". Finally, the hamming distance between the bit vector from the testing instance and each class is calculated, and this testing sample is assigned to the class with smallest hamming distance.

3.3 Classifier Ensemble

Suppose we have an instances $\mathbf{x}$, where $\mathbf{x}$ is a d-dimensional feature vector. Let $\omega_1, \omega_2, \cdots, \omega_M$ be M categories, and $\alpha_1, \alpha_2, \cdots, \alpha_M$ be a finite set of possible actions. Suppose we have totally N classifiers, namely $c_1, c_2, \cdots, c_N$. Each classifier will generate a posterior probability $P_{c_n}(\omega_j|x)$ for $\mathbf{x}$. Here, we define a loss function $\lambda(\alpha_i|\omega_j)$ which describes the loss occurred for taking action α_i when the state of nature is ω_j. Obviously, we can get a set of posterior probabilities used for classification generated by different classifiers as:

$$P_{c_1}(\omega_j|x), P_{c_2}(\omega_j|x), \cdots, P_{c_n}(\omega_j|x).$$

For each probability function, the expected loss associated with taking action α_i is defined in Eq. (1).

$$R_{c_n}(\alpha_i|x) = \sum_{j=1}^{M} \lambda(\alpha_i|\omega_j) P_{c_n}(\omega_j|x) \tag{1}$$

Then, as a classification problem, a zero-one loss function is defined as:

$$\lambda(\alpha_i|\omega_j) = \begin{cases} 0 & i = j \\ 1 & i \neq j \end{cases}, \ where\, i,j = 1, 2, \cdots, M \tag{2}$$

Using these definitions, for each classifier, the condition risk for category $\boldsymbol{\omega}_j$ is defined in Eq. (3). R needs to be minimized to achieve the best performance for a certain classifier c_n using Eq. (4) as follows.

$$R_{c_n}(\alpha_i|x) = \sum_{j \neq i} P_{c_n}(\omega_j|x) = 1 - P_{c_n}(\omega_i|x) \tag{3}$$

$$R_{c_n} = \int_{x \in \Omega} R_{c_n}(\alpha|x)p(x)dx = \int_{x \in \Omega} [1 - P_{c_n}(\omega|x)]p(x)dx \tag{4}$$

Considering N different classifiers, most previous fusion methods use a certain algorithm to fuse different $P_{c_n}(\omega_j|x)$ for M categories. For example, using the weighted combination rules, we can generate a combined posterior and a new conditional risk R using Eqs. (5) and (6).

$$P_{fusion}(\omega_j|x) = \sum_{n=1}^{N} w_n P_{C_n}(\omega_j|x) \tag{5}$$

$$\begin{aligned} R_{fusion} &= \int_{x \in \Omega} \left[1 - P_{fusion}(\omega|x)\right]p(x)dx \\ &= \int_{x \in \Omega_1} \left[1 - P_{fusion}(\omega_1|x)\right]p(x)dx + \int_{x \in \Omega_2} \left[1 - P_{fusion}(\omega_2|x)\right]p(x)dx \\ &\quad + \cdots + \int_{x \in \Omega_M} \left[1 - P_{fusion}(\omega_M|x)\right]p(x)dx, \\ &\quad \textit{where } \Omega_1 \cup \Omega_2 \cup \cdots \cup \Omega_M = \Omega \textit{ and } \Omega_i \cap \Omega_j = \phi \\ &\quad (i \neq j, i, j = 1, 2, \cdots, M) \end{aligned} \tag{6}$$

As discussed in Sect. 2, the issue of earlier ensemble models is that we can only fuse classifiers performing well in all categories while integrating a relatively bad classifier may lead to even worse results and eventually reduces the performance in most of the time. However, as a bad guy with a good point, a relatively bad classifier may outperform a good classifier for a certain class. Thus, the proposed framework can still use it to enhance the classification result even though it is not a good choice for all the other classes. We did this by splitting a classifier is split into different "judgers", with each judger working independently to determine the label of a testing instance. We can thus find the good point in a bad classifier and the conditional risk can be reduced by using different posteriors for different classes, as shown in Eq. (7).

$$\begin{aligned} R_{\min} = &\int_{x \in \Omega_1} [1 - P_{\max}(\omega_1|x)]p(x)dx + \int_{x \in \Omega_2} [1 - P_{\max}(\omega_2|x)]p(x)dx \\ &+ \cdots + \int_{x \in \Omega_M} [1 - P_{\max}(\omega_M|x)]p(x)dx \end{aligned} \tag{7}$$

3.4 Generation of Judgers

In order to generate judgers, our classifier fusion model firstly split a dataset into three parts including a training, a validation, and a testing dataset. The classification models are then trained using the training dataset, followed by the calculation of precision and recall on the corresponding validation dataset. Here, precision is defined as a positive judger and recall rate is denoted as a negative judger, where *TP* stands for the true positive value, and *FP* and *FN* are the false positive and false negative values, respectively as follows:

$$J_{pos} = precision = \frac{TP}{TP+FP}$$
$$J_{neg} = recall = \frac{TP}{TP+FN}$$

Suppose there are M classes in a certain dataset. For one type of features f_l ($l \in [1, L]$, L is the number of feature descriptors, which is two in this paper), based on the classification results on the validation dataset, $2 \times M$ judgers will be generated for a certain classifier c_n ($n \in [1, N]$, N is the total number of the classification models built on one type of features) as follows:

$$J^{n,l}_{pos_1}, J^{n,l}_{pos_2}, J^{n,l}_{pos_3}, \cdots J^{n,l}_{pos_m} \cdots, J^{n,l}_{pos_M}$$
$$J^{n,l}_{neg_1}, J^{n,l}_{neg_2}, J^{n,l}_{neg_3}, \cdots J^{n,l}_{neg_m} \cdots, J^{n,l}_{neg_M}$$

Here, $J^{n,l}_{pos_m}$ is a positive judger generated by classifier c_n using feature f_l for the class ω_m (($m \in [1, M]$, M is the total number of categories). Correspondingly, $J^{n,l}_{neg_m}$ is a negative judger. If $J^{n,l}_{pos_m}$ is high, it indicates that this judger is relatively accurate. Accordingly, if it judges a testing instance as in class ω_m, it is highly likely that this judgment is correct. On the other hand, if $J^{n,l}_{neg_m}$ is high, it can be considered as a good negative judger since if classifier c_n does not label a testing instance as class ω_m, the ground truth of the instance is not likely to be ω_m. These judgers form the committee to give the final classification results.

In summary, suppose there are totally N classifiers and L types of features fed for each classification model, the total number of judgers is *2MNL*. As an instance, for the UCF11 dataset [30] used in this work, there are 11 classes. Considering all the three types of classifiers introduced on two kinds of features, the total number of judgers generated would be 132, which is equal to $2 \times 11 \times 3 \times 2$.

3.5 The Classifiers Fusion Model

After both positive and negative judgers are generated, the next important issue is how to fuse the outputs from different judgers to draw the final conclusion. In order to assemble these judgers, a novel classifier ensemble framework is proposed as follows:

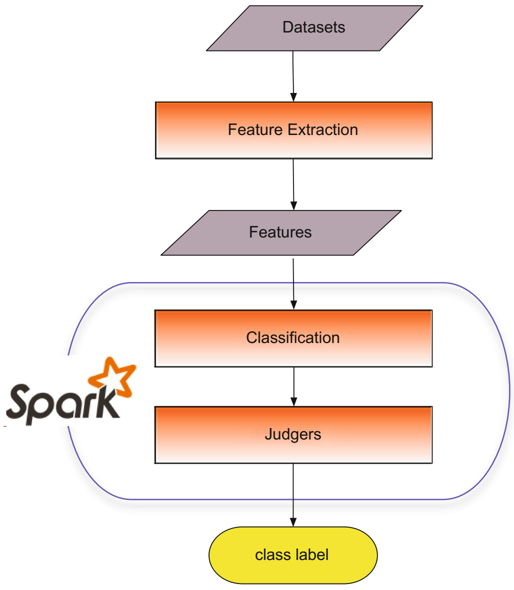


Fig. 1. The proposed classifier ensemble framework.

As shown in Fig. 1, all the judgers generated are used to build a novel classifier fusion model. For a testing instance **x**, each classifier will assign **x** a serial of positive and negative judgers on each feature space. Suppose we have the following list of judgers:

$$J^{11}_{pos_1}, J^{11}_{neg_1}, J^{11}_{pos_2}, J^{11}_{neg_2}, \cdots, J^{21}_{pos_1}, \cdots J^{31}_{pos_1}, \cdots J^{32}_{pos_M}, J^{32}_{neg_M}$$

Here, the positive judgers are first used and will be re-ranked by their accuracies. When the highest positive judger ranks the first and assigns **x** as class ω_m, **x** will be determined as class ω_m. For example, for a testing instance, if $J^{32}_{pos_1}$ is the largest positive judger with the highest accuracy, that testing instance will be classified as class 1.

In the next step, the highest positive judger will be compared with the largest negative judger for the same class (i.e., $J^{n,l}_{neg_1}$ in the previous example). Take the same example, if the value of $J^{21}_{neg_1}$ is larger than $J^{32}_{pos_1}$(meaning that the negative judger dominates the classification), **x** won't be assigned to class 1 because there exists a larger negative judger. That is to say even if the largest positive judger assigns a testing instance to class ω_m, it will be skipped and the second largest positive judger followed will be considered and compared with the corresponding largest negative judger. A similar process continues until a positive judger assigns **x** to a class without the corresponding largest negative judger rejecting it. The results of this particular testing instance are showed as follows:

$$J^{32}_{pos_1}, J^{12}_{pos_3}, J^{31}_{pos_8}, J^{22}_{pos_2}, \cdots$$
$$J^{32}_{pos_1} < J^{21}_{neg_1}$$
$$J^{12}_{pos_3} > \max J^{n,j}_{neg_3}$$

In this example, our proposed model assigns **x** to class 3. This procedure is applied to all the testing instances. In very rare cases, however, it should be noticed that if all the corresponding largest negative judgers reject the classification results from the positive judgers, the testing instance will be assigned back to the decision of the highest positive judger at the beginning.

3.6 Proposed Apache Spark Cluster

Nowadays, a number of large-scale data processing frameworks are turning towards generalized MapReduce frameworks after Apache Hadoop$^{\text{TM}}$ is released. Among them, Spark is an open source big data processing framework advertised as extremely fast cluster computing and increases the capability of conventional MapReduce use-cases. It is a fast and general engine for big data processing and has been deployed for many popular large-scale systems. With cores complemented by a set of higher-level libraries including Spark Streaming, SparkSQL for NoSQL database, GraphX for graphs computation, and Mllib for machine learning.

Based on Spark, an efficient system for classifier fusion of large-scale data is built as shown in Fig. 2. Spark Core is the foundation of the overall project which provides the distributed task dispatching, scheduling, and basic I/O functionalities. Currently,

Fig. 2. Our spark cluster.

our Spark cluster contains four boxes with one master node and three slave nodes. Each node is setup to instantiate 2 workers with 4 GB of memory and 1 TB of storage.

The main abstraction Spark provided is a Resilient Distributed Dataset (RDD) which is an immutable fault-tolerant, distributed collection of objects that can be operated in parallel. An RDD can contain any type of objects and is created by loading an external dataset or distributing a collection from the driver program. In addition to the only two operations in MapReduce, Spark provides many other operations called transformations such as map, sample, groupByKey, reduceByKey, union, join, sort, mapValues, and partionBy. The above operations can be used either stand-alone or in combination to run in a single data pipeline use case. Currently, Spark is originally written in Scala and now fully supports Java; while it also supports Python unstably. In this paper, our codes are all written in Java.

In details, we first read the keys (sample ID) and values (scores from different classifiers for different features) as follows to build a very efficient multimedia large-scale data classification model using Spark:

$$
\begin{aligned}
&Key = sample_1,\ Value = (s_1^{c_1,f_1,\omega_1}, s_1^{c_1,f_1,\omega_2} \cdots s_1^{c_2,f_1,\omega_1} \cdots s_1^{c_N,f_L,\omega_M})\\
&Key = sample_2,\ Value = (s_2^{c_1,f_1,\omega_1}, s_2^{c_1,f_1,\omega_2} \cdots s_2^{c_2,f_1,\omega_1} \cdots s_2^{c_N,f_L,\omega_M})\\
&Key = sample_3,\ Value = (s_3^{c_1,f_1,\omega_1}, s_3^{c_1,f_1,\omega_2} \cdots s_3^{c_2,f_1,\omega_1} \cdots s_3^{c_N,f_L,\omega_M})\\
&\ldots\\
&Key = sample_P, Value = (s_P^{c_1,f_1,\omega_1}, s_P^{c_1,f_1,\omega_2} \cdots s_P^{c_2,f_1,\omega_1} \cdots s_P^{c_N,f_L,\omega_M})
\end{aligned}
$$

Here, the meanings and notations are the same as introduced in previous sections and the output values are the classification results. To easily compare our performance with other existing approaches, two popular medium-sized video datasets are used in the experiments. Nevertheless, our Spark cluster is able to handle large-scale multimedia datasets easily. For a larger dataset, more key-value pairs will be created and thus the system can help more in terms of efficiency in comparison to classical classifier fusion models.

4 Experiments and Result Analyses

To test the efficiency of our classifier ensemble framework, two popular and widely accepted benchmark multimedia datasets in the field of human action recognition are used in the experiments. These two datasets are the KTH dataset [46] and the UCF11 dataset [47].

In the two experiments on KTH and UCF11, the 25-fold cross validation is adopted. Three classifiers introduced in Sect. 3.2 are used, namely SVM, SRC, and HDC; while both SIFT and STIP features are used.

4.1 Results on the KTH Dataset

The KTH dataset includes 6 different human actions (i.e., boxing, hand clapping, waving, jogging, running, and walking) from 25 actors in 4 kinds of scenarios

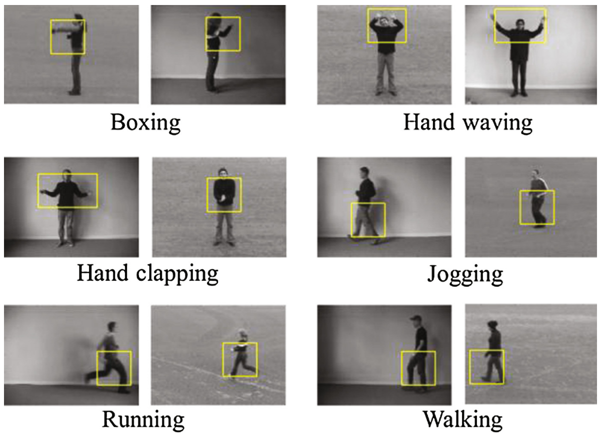


Fig. 3. Sample frames from the KTH dataset.

(namely indoors, outdoors, outdoors with scale variation, and outdoors with different clothes). Thus, there are 600 video sequences in total. All videos are in the "avi" format and were recorded in a controlled setting with slight camera motion and a simple background as shown in Fig. 3.

Table 1 shows the confusion matrix of our results. To fully evaluate our model, our fusion strategy is compared with 6 other fusion strategies in terms of accuracy and the results are given in Table 2. The arithmetic mean and geometric mean represent the sum and product of the scores. Two ways of hybrid means are also tested. One is first to calculate the arithmetic mean scores among different classifiers based on the same kind of features and then to compute the geometric mean scores between different kinds of features; the other kind is to do the opposite.

Table 1. The confusion matrix of six action categories in the KTH dataset.

	Box	Clap	Wave	Jog	Run	Walk
Box	96	3	0	0	0	1
Clap	0	99	0	0	0	1
Wave	1	5	94	0	0	0
Jog	0	0	0	88	11	1
Run	0	0	0	0	100	0
Walk	0	0	0	0	0	100

Table 2. Comparison of our classifier ensemble framework and other fusion algorithms on the KTH dataset.

Algorithm	Average precision
Arithmetic mean	90.7%
Geometric mean	90.0%
Hybrid mean (Arithmetic for different features)	90.7%
Hybrid mean (Geometric for different features)	90.3%
Linear regression on SIFT	90.5%
Linear regression on STIP	91.2%
The proposed strategy	**95.2%**

In addition, our classifier ensemble model is compared to 4 other published frameworks. Table 3 compares the accuracies among them. We also split our experiment by first using only SIFT and then using only STIP as described in Sect. 3. As can be clearly seen from the comparison results, our proposed model performs better the other ones.

Table 3. Comparison of overall average precision of our method and state-of-the-art methods on the KTH dataset.

Method	Average precision
Schuldt et al. [46]	71.5%
Dollar et al. [48]	80.7%
Yin et al. [49]	82.0%
Niebles et al. [50]	91.3%
Our work on SIFT	**93.7%**
Our work on STIP	**95.3%**
Our work on both features	**96.2%**

4.2 Results on the UCF11 Dataset

The UCF11 dataset which also known as "YouTube Action Dataset" is more challenging than the KTH dataset, since it contains realistic actions, camera motions, and complicated backgrounds. There are eleven action categories, namely basketball shooting, biking/cycling, diving, golf swinging, horseback riding, soccer juggling, swinging, tennis swinging, trampoline jumping, volleyball spiking, and walking with a dog. For each category, the videos are grouped into 25 groups with more than 4 action clips in it. Some sample frames are given in Fig. 4.

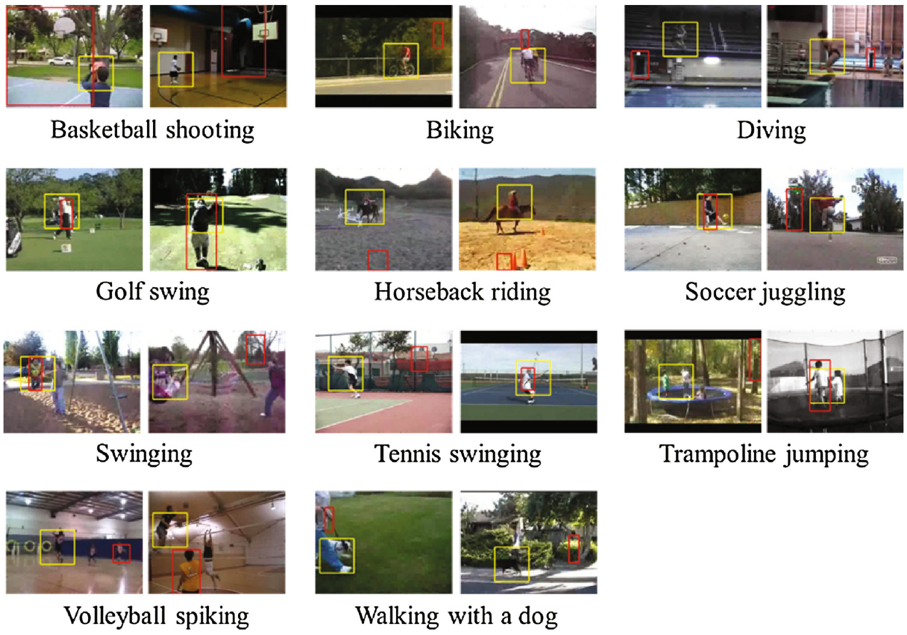


Fig. 4. Sample frames from the UCF dataset.

Similar to the experimental steps in Sect. 4.1, we compare our work with 6 other fusion strategies in terms of accuracy and the results are given in Table 4. Meanwhile, we also compare our results with 4 other state-of-the-art models and the results are shown in Table 5. Table 5 also shows the advantage of the proposed model which can fuse different kinds of features to achieve a better performance than the other methods, though the result by only one kind of feature may not able to always outperform other methods.

Table 4. Comparison of our ensemble model and other fusion algorithms for the UCF11 dataset.

Algorithm	Average precision
Arithmetic mean	74.91%
Geometric mean	74.82%
Hybrid mean (Arithmetic for different features)	75.27%
Hybrid mean (Geometric for different features)	75.09%
Linear regression on SIFT	77.82%
Linear regression on STIP	75.91%
The proposed strategy	**80.3%**

Table 5. Comparison of overall average precision of our method and state-of-the-art methods for the UCF11 dataset.

Method	Average precision
Chen et al. [51]	67.5%
Perez et al. [52]	68.9%
Liu et al. [53]	71.2%
Mota et al. [54]	75.4%
Multi-classifier on SIFT	**69.1%**
Multi-classifier on STIP	**74.8%**
Our work on both features	**80.3%**

5 Conclusions

Recently, ensemble learning models have received a lot of attentions with the attempt to take advantages of multiple useful classifiers. In this paper, a novel classifier ensemble framework based on judgers is proposed to fuse the classification results generated from different classifiers and features. The proposed framework is built on an Apache Spark cluster and applied to two different human action video datasets as a proof-of-concept. The experimental results show that our proposed framework outperforms several existing state-of-the-art classification approaches.

Although these two datasets are medium-sized, the proposed framework is capable of handling big datasets as it is built on Spark. Thus, the proposed framework can be easily extended to work with more classifiers and features for other multi-class and multi-feature classification problems. Theoretically, the more classifiers and features included, the more judgers will be generated correspondingly, which can potentially lead to better classification results.

Acknowledgment. For Dr. Shu-Ching Chen, this research work is partially supported by DHS's VACCINE Center under Award Number 2009-ST-061-CI0001 and NSF HRD-0833093, HRD-1547798, CNS-1126619, and CNS-1461926.

References

1. Lin, L., Shyu, M.-L.: Weighted association rule mining for video semantic detection. Int. J. Multimed. Data Eng. Manage. **1**(1), 37–54 (2010)
2. Zhu, Q., Lin, L., Shyu, M.-L., Chen, S.-C.: Feature selection using correlation and reliability based scoring metric for video semantic detection. In: Proceedings of the Fourth IEEE International Conference on Semantic Computing, pp. 462–469, September 2010
3. Shyu, M.-L., Quirino, T., Xie, Z., Chen, S.-C., Chang, L.: Network intrusion detection through adaptive sub-eigenspace modeling in multiagent systems. ACM Trans. Auton. Adaptive Syst. **2**(3), 9:1–9:37 (2007). Article No. 9
4. Shyu, M.-L., Chen, S.-C., Chen, M., Zhang, C.: A unified framework for image database clustering and content-based retrieval. In: Proceedings of the 2nd ACM International Workshop on Multimedia Databases, pp. 19–27, New York, NY, USA (2004)

5. Chen, S.-C., Shyu, M.-L., Zhang, C.: An intelligent framework for spatio-temporal vehicle tracking. In: Proceedings of the 4th IEEE International Conference on Intelligent Transportation Systems, pp. 213–218, August 2001
6. Chen, S.-C., Shyu, M.-L., Zhang, C., Kashyap, R.L.: Identifying overlapped objects for video indexing and modeling in multimedia database systems. Int. J. Artif. Intell. Tools **10** (4), 715–734 (2001)
7. Chen, S.-C., Shyu, M.-L., Zhang, C.: Innovative shot boundary detection for video indexing. In: Deb, S. (ed.) Video Data Management and Information Retrieval. Idea Group Publishing, pp. 217–236 (2005)
8. Shyu, M.-L., Haruechaiyasak, C., Chen, S.-C., Zhao, N.: Collaborative filtering by mining association rules from user access sequences. In: Proceedings of the International Workshop on Challenges in Web Information Retrieval and Integration, pp. 128–135, April 2005
9. Chen, S.-C., Rubin, S., Shyu, M.-L., Zhang, C.: A dynamic user concept pattern learning framework for content-based image retrieval. IEEE Trans. Syst. Man Cybern. Part C Appl. Rev. **36**(6), 772–783 (2006)
10. Shyu, M.-L., Xie, Z., Chen, M., Chen, S.C.: Video semantic event/concept detection using a subspace-based multimedia data mining framework. IEEE Trans. Multimed. **10**(2), 252–259 (2008)
11. Lin, L., Ravitz, G., Shyu, M.-L., Chen, S.-C.: Effective feature space reduction with imbalanced data for semantic concept detection. In: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, pp. 262–269, June 2008
12. Shyu, M.-L., Chen, S.-C., Kashyap, R.: Generalized affinity-based association rule mining for multimedia database queries. Knowl. Inf. Syst. (KAIS) Int. J. **3**(3), 319–337 (2001)
13. Huang, X., Chen, S.-C., Shyu, M.-L., Zhang, C.: User concept pattern discovery using relevance feedback and multiple instance learning for content-based image retrieval. In: Proceedings of the Third International Workshop on Multimedia Data Mining, in conjunction with the 8th ACM International Conference on Knowledge Discovery & Data Mining, pp. 100–108, July 2002
14. Li, X., Chen, S.-C., Shyu, M.-L., Furht, B.: Image retrieval by color, texture, and spatial information. In: Proceedings of the 8th International Conference on Distributed Multimedia Systems, pp. 152–159, September 2002
15. Lin, L., Ravitz, G., Shyu, M.-L., Chen, S.-C.: Video semantic concept discovery using multimodal-based association classification. In: Proceedings of the IEEE International Conference on Multimedia & Expo, pp. 859–862, July 2007
16. Shyu, M.-L., Chen, S.-C., Chen, M., Zhang, C., Sarinnapakorn, K.: Image database retrieval utilizing affinity relationships. In: Proceedings of the 1st ACM International Workshop on Multimedia Databases, pp. 78–85, New York, NY, USA (2003)
17. Shyu, M.-L., Haruechaiyasak, C., Chen, S.-C.: Category cluster discovery from distributed WWW directories. Inf. Sci. **155**(3), 181–197 (2003)
18. Perez, E.A., Mota, V.F., Maciel, L.M., Sad, D., Vieira, M.B.: Combining gradient histograms using orientation tensors for human action recognition. In: Proceedings of the International Conference on Pattern Recognition, pp. 3460–3463 (2012)
19. Yin, B., Ruiz, N., Chen, F., Ambikairajah, E.: Investigating speech features and automatic measurement of cognitive load. In: Proceedings of the IEEE 10th Workshop on Multimedia Signal Processing, pp. 988–993, October 2008
20. Yan, Y., Pouyanfar, S., Tian, H., Guan, S., Ha, H.-Y., Chen, S.-C., Shyu, M.-L., Hamid, S.: Domain knowledge assisted data processing for Florida Public Hurricane Loss Model. In: Proceedings of the 17th IEEE International Conference on Information Reuse and Integration (IRI), pp. 441–447, Pittsburgh, PA, USA, July 2016

21. Liu, D., Yan, Y., Shyu, M.-L., Zhao, G., Chen, M.: Spatio-temporal analysis for human action detection and recognition in uncontrolled environments. Int. J. Multimed. Data Eng. Manage. **6**(1), 1–18 (2015)
22. Yan, Y., Shyu, M.-L.: Enhancing rare class mining in multimedia big data by concept correlation. In: Proceedings of the IEEE International Symposium on Multimedia (ISM), pp. 281–286, San Jose, CA, USA, December 2016
23. Yan, Y., Zhu, Q., Shyu, M.-L., Chen, S.-C.: A classifier ensemble framework for multimedia big data classification. In: Proceedings of the 17th IEEE International Conference on Information Reuse and Integration (IRI), pp. 615–622, Pittsburgh, PA, USA, July 2016
24. Yan, Y., Shyu, M.-L., Zhu, Q.: Supporting semantic concept retrieval with negative correlations in a multimedia big data mining system. Int. J. Semant. Comput. (IJSC) **10**(2), 247–268 (2016)
25. Yan, Y., Chen, M., Shyu, M.-L., Chen, S.-C.: Deep learning for imbalanced multimedia data classification. In: Proceedings of the 2015 IEEE International Symposium on Multimedia (ISM), pp. 483–488, Miami, FL, December 2015
26. Yan, Y., Shyu, M.-L., Zhu, Q.: Negative correlation discovery for big multimedia data semantic concept mining and retrieval. In: Proceedings of the 2016 IEEE Tenth International Conference on Semantic Computing, pp. 55–62, Laguna Hills, CA, February 2016
27. Duin, R.P.W., Tax, D.M.J.: Experiments with classifier combining rules. In: Kittler, J., Fabio Roli (Eds.) Proceedings of the 1st International Workshop on Multiple Classifier Systems (2000)
28. Yan, Y., Liu, Y., Shyu, M.-L., Chen, M.: Utilizing concept correlations for effective imbalanced data classification. In: Proceedings of the 15th IEEE International Conference on Information Reuse and Integration, pp. 561–568, August 2014
29. Merler, M., Huang, B., Xie, L., Gang, H., Natsev, A.: Semantic model vectors for complex video event recognition. IEEE Trans. Multimed. **14**(1), 88–101 (2012)
30. Zhu, Q., Shyu, M.-L.: Sparse linear integration of content and context modalities for semantic concept retrieval. IEEE Trans. Emerg. Topics Comput. **3**(2), 152–160 (2015)
31. Duin, R.P.W.: The combining classifier: to train or not to train? In: Proceedings of the 16th International Conference on Pattern Recognition, vol. 2, pp. 765–770 (2002)
32. Ashfaq, A.B., Javed, M., Khayam, S.A., Radha, H.: An information-theoretic combining method for multi-classifier anomaly detection systems. In: Proceedings of the IEEE International Conference on Communications, pp. 1–5, May 2010
33. Meng, T., Shyu, M.-L., Lin, L.: Multimodal information integration and fusion for histology image classification. Int. J. Multimed. Data Eng. Manage. **2**(2), 54–70 (2011)
34. Liu, N., Dellandréa, E., Chen, L., Zhu, C., Zhang, Y., Bichot, C.-E., Bres, S., Tellez, B.: Multimodal recognition of visual concepts using histograms of textual concepts and selective weighted late fusion scheme. Comput. Vis. Image Underst. **117**(5), 493–512 (2013)
35. Kittler, J., Hatef, M., Duin, R.P.W., Matas, J.: On combining classifiers. IEEE Trans. Pattern Anal. Mach. Intell. **20**(3), 226–239 (1998)
36. Kuncheva, L.I.: Combining Pattern Classifiers: Methods and Algorithms. Wiley-Interscience, Hoboken (2004)
37. Meng, T., Shyu, M.-L.: Leveraging concept association network for multimedia rare concept mining and retrieval. In: Proceedings of the 2012 IEEE International Conference on Multimedia and Expo, pp. 860–865, July 2012
38. Xu, L., Krzyzak, A., Suen, C.Y.: Methods of combining multiple classifiers and their applications to handwriting recognition. IEEE Trans. Syst. Man Cybern. **22**(3), 418–435 (1992)
39. Al-Ani, A., Deriche, M.: A new technique for combining multiple classifiers using the Dempster-Shafer theory of evidence. J. Artif. Intell. Res. **17**, 333–361 (2002)

40. Liu, D., Shyu, M.-L.: Effective moving object detection and retrieval via integrating spatial-temporal multimedia information. In: Proceedings of the IEEE International Symposium on Multimedia, pp. 364–371, December 2012
41. Chen, L.-C., Hsieh, J.-W., Yan, Y., Chen, D.-Y.: Vehicle make and model recognition using sparse representation and symmetrical SURFs. Pattern Recogn. **48**(6), 1979–1998 (2015)
42. Laptev, I.: On space-time interest points. Int. J. Comput. Vis. **64**(2–3), 107–123 (2005)
43. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. Int. J. Comput. Vis. **60**(2), 91–110 (2004)
44. Mairal, J., Bach, F., Ponce, J., Sapiro, G.: Online learning for matrix factorization and sparse coding. J. Mach. Learn. Res. **11**, 19–60 (2010)
45. Yan, Y., Hsieh, J.-W., Chiang, H.-F., Cheng, S.-C., Chen, D.-Y.: PLSA-based sparse representation for object classification. In: Proceedings of the 22nd International Conference on Pattern Recognition, pp. 1295–1300, August 2014
46. Schuldt, C., Laptev, I., Caputo, B.: Recognizing human actions: a local SVM approach. In: Proceedings of the 17th International Conference on Pattern Recognition, vol. 3, pp. 32–36, August 2004
47. Liu, J., Yang, Y., Shah, M.: Learning semantic visual vocabularies using diffusion distance. In: Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition (CVPR), pp. 461–468, June 2009
48. Dollar, P., Rabaud, V., Cottrell, G., Belongie, S.: Behavior recognition via sparse spatio-temporal features. In: Proceedings of the International Conference on Computer Vision Workshop Visual Surveillance Performance Evaluation Tracking Surveillance, pp. 65–72, October 2005
49. Yin, J., Meng, Y.: Human activity recognition in video using a hierarchical probabilistic latent model. In: Proceedings of the 17th International Conference on Pattern Recognition, pp. 15–20 (2010)
50. Niebles, J.C., Chen, C.-W., Fei-Fei, L.: Modeling temporal structure of decomposable motion segments for activity classification. In: Proceedings of the European Conference on Computer Vision, pp. 1–14 (2010)
51. Chen, X., Liu, J., Liu, H.: Will scene information help realistic action recognition? In: Proceedings of the 10th World Congress on Intelligent Control and Automation (WCICA), pp. 4532–4535, July 2012
52. Liu, J., Luo, J., Shah, M.: Recognizing realistic actions from videos 'in the wild'. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1996–2003, June 2009
53. Mota, V.F., Souza, J.I.C., Araujo, A.D.A., Bernardes, V.M.: Combining orientation tensors for human action recognition. In: Proceedings of the 26th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI), pp. 328–333, August 2013

Employing Graph Databases as a Standardization Model for Addressing Heterogeneity and Integration

Dippy Aggarwal(✉) and Karen C. Davis(✉)

University of Cincinnati, Cincinnati, OH 45220, USA
aggarwdy@mail.uc.edu, karen.davis@uc.edu

Abstract. The advent of big data and NoSQL data stores has led to the proliferation of data models exacerbating the challenges of information integration and exchange. It would be useful to have an approach that allows leveraging both schema-based and schema-less data stores. We present a graph-based solution that attempts to bridge the gap between different data stores using a homogeneous representation. As the first contribution, we present and demonstrate a mapping approach to transform schemas into a homogeneous graph representation. We demonstrate our approach over relational and RDF schemas but the framework is extensible to allow further integration of additional data stores. The second contribution is a schema merging algorithm over property graphs. We focus on providing a modular framework that can be extended and optimized using different schema matching and merging algorithms.

Keywords: Schema integration · Graph databases · Schema mapping · Neo4j

1 Introduction

Recently, there has been a shift in the volume of data generated, diversity in data forms (structured, semi-structured and unstructured) and an unprecedented rate at which data is produced. The data possessing these characteristics is termed as big data and the field is forecasted to grow at a 26.4% compound annual growth rate through 2018, according to a report released by Intelligent Data Corporation [33]. The need for faster analysis over big data with current storage and computation power poses a major challenge for enterprise data infrastructures. Two alternatives to handle analysis over this newer form of data are: (a) scale-up (adding CPU power, memory to a single machine), and (b) scale-out (adding more machines in the system creating a cluster). The main advantage of scaling out vs. scaling up is that more work can be done by parallelizing the workload across distributed machines. Many existing relational database systems are designed to perform efficiently on single-machines. It should not be misconstrued that relational systems cannot scale-out at all. They can, but they

S.H. Rubin and T. Bouabana-Tebibel (eds.), *Quality Software Through Reuse and Integration*, Advances in Intelligent Systems and Computing 561,
DOI 10.1007/978-3-319-56157-8_6

lose the features that they are primarily designed such as ACID compliance, for example [34].

NoSQL (Not Only SQL) describes an emerging class of storage models designed for scalable database systems. NoSQL data stores advocate new and relaxed forms of data storage that do not require a schema to be enforced for the underlying data. Instead, a schema is identified and generated at the application side when the data is read from the system. This concept of postponing schema definition to a later point has enabled NoSQL storage models to be applied in many real-world use-cases. However, the popularity has also created a notion that schemaless data management techniques are more suitable for solving emerging data problems than schema-based structures.

While NoSQL data stores offer interesting and novel solutions for managing big data, they are also not a panacea for all data management related scenarios. In scenarios that need query optimization, data governance, and integrity, schema-based stores offer a better solution. Furthermore, a large amount of enterprise data still resides in relational databases. The greater scalability of NoSQL databases over relational databases comes at a price. Most NoSQL systems compromise certain features, such as strong consistency, to achieve efficiency over other critical features of performance and availability. Organizations such as Facebook and Hadapt who have widely embraced big data technologies also choose a data store on a use-case basis as opposed to leveraging a single big data storage technique for all their applications and data storage requirements [35]. More recently, a number of SQL implementations have also emerged that are built over platforms and programming models such as MapReduce that support big data [36–41].

Another field that is growing rapidly is semantic web and linked data technologies. Linked data builds upon the existing web and offers the idea of annotating the web data (and not just the documents) [42] using global identifiers and linking them. The data is published and organized using RDF (Resource Description Framework), which is based on a subject-object-predicate framework. RDF schema specifications and modeling concepts differ from relational databases and NoSQL data models, thus supporting the need and demand for creating schema and data integration solutions. Hitzler et al. [42] identify linked data as a part of the big data landscape.

These recent developments indicate two ideas: (1) the significance and prevalence of structured data in the enterprise world, and (2) the unique advantages possessed by different classes of data stores. In order to reap benefits from all of them, it is important to bring them together under a homogeneous model. This would serve two purposes: (1) offer more complete knowledge by combining data stored in isolated sources, and (2) facilitate harnessing value from each of the data stores (schema-based and schema-less), thus making them complementary and not competitive solutions.

In this paper, we address this need by adopting graphs as a means towards standardization and integration of different data stores, thus handling the variety characteristic of big data. Our selection of a graph model is based on the following observations:

1. Graph databases are a NoSQL data storage model and thus support the big data processing framework [50].
2. Graphs provide a simple and flexible abstraction for modeling artifacts of different kinds in the form of nodes and edges.
3. Graph databases are attracting significant attention and interest in the past few years as highlighted from the Google Trends analysis shown in Fig. 1. The values are normalized representing the highest value in the chart as 100% and the x-axis labels are marked in two-year time intervals.
4. The graph model adopted in our work, Neo4j, possesses a query language called Cypher and allows programmatic access using API.

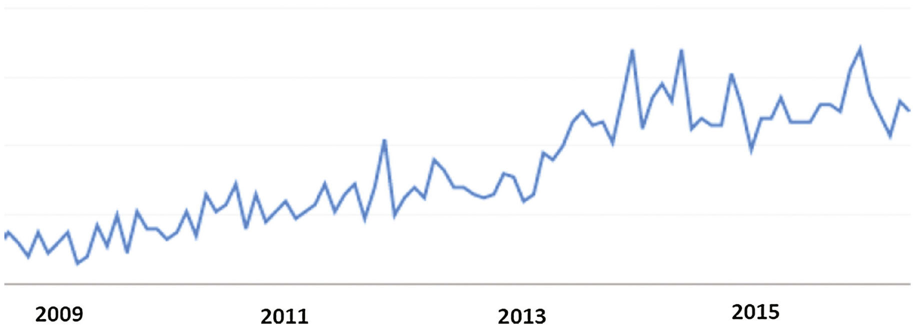


Fig. 1. Trend of web search interest for graph databases [28]

The main contributions of this paper are as follows:

1. A concept-preserving, integrated graph model that addresses the model heterogeneity and variety dimension of the big data landscape.
2. A software-oriented, automated approach to transform relational and RDF schemas into a graph database.
3. A proof-of-concept that illustrates the potential of graph-based solutions towards addressing diversity in data representations.
4. A framework accompanied by a proof-of-concept for schema merging over property graphs.

The rest of the paper is organized as follows. Section 2 presents an overview of the concepts and terms that are used frequently in the paper. These include a discussion of property graphs and Neo4j graph database in particular, and relational and RDF data models as our native models of interest. Section 3 describes our transformation rules for converting a relational schema to a property graph. We leverage the approach proposed by Bouhali et al. [51] for converting RDF to a property graph representation and extend it to support additional models.

Next, we present a proof-of-concept to illustrate the implementation of our transformation rules. We consider schema excerpts for relational and RDF models and show their corresponding property graphs generated using the transformation rules. Section 4 introduces our architecture and the motivation behind mapping non-graph based schemas to a property graph. The evaluation issues for schema mapping are reported in Sect. 5. We next present our approach towards schema integration over property graphs in Sect. 6. The challenges and our proposed solution towards resolving them are illustrated using an example. Section 7 presents the schema integration algorithm. In Sect. 8, we use two case-studies to evaluate our approach for schema integration. Section 9 addresses the related work in graph-based integration and transformations while Sect. 10 offers conclusions and future work.

2 Background

In this section, we overview the concepts that form a foundation for our research. These include relational and RDF schemas that serve as input schemas. There are many graph based models; we consider the property graph as our model of interest and introduce it here briefly. We leverage Neo4j [52] in our work.

2.1 RDF

RDF stands for Resource Description Framework. It refers to the model and RDF schema is commonly abbreviated as RDFS. RDF allows annotating web resources with semantics. The resources and semantic information is represented in the form of classes and properties which form two core concepts of an RDF schema.

The notion of classes and objects in RDFS differ from the similar concepts that exist in conventional, object-oriented systems [53]. In many systems, classes contain a set of properties and each of the class instances possesses those properties. However, in RDF, properties are described in terms of classes to which they apply. These classes are referred to as *domain* in RDF schema, and similarly, the values that a certain property can hold is described using *range*. For example, we could define a property *member* that has domain *Group* and its range would be *Person*. Nejdl et al. [43] and W3C specification [44] offer a summary of RDF classes and properties.

2.2 Relational Schema

A relational schema is described as a set of relations which consists of a set of attributes and constraints. The relations are connected by different types of relationships with cardinality constraints. There are two major types of constraints: entity integrity and referential integrity. The entity integrity constraint states that every relation has a set of attributes, termed *primary key*, that uniquely identifies each of the tuples in the relation. The primary key attribute(s) may not be null. The referential integrity constraint applies to a pair of relations that are associated with each other. Having this constraint signifies that every value of one attribute in one of the participating relations comes from the set of values of a primary key attribute in the other relation.

2.3 Neo4j and Property Graphs

Graphs provide a simple and flexible abstraction for modeling artifacts of different kinds in the form of nodes and edges. The graph model adopted in our work, Neo4j [52] supports automation and has a query language [45]. The database is available for download for free and there is vast technical support and a large user base. Neo4j has also been ranked as the most popular graph database by db-engines.com [46].

The graph database in our work, Neo4j [52], organizes its data as a labelled property graph in which the nodes and relationships possess properties and can be annotated with labels. This allows augmenting the graph nodes and edges with semantics. The concept of properties is analogous to attributes in traditional conceptual models such as the Entity-Relationship Model. In property graphs, properties are key-value pairs. Figure 2 presents an example of data modeling using a property graph.

Some key points to note in Fig. 2 are as follows: (1) the labels *Person, Book*, and *Author* are depicted in rectangles over the nodes, (2) a node can possess more than one label, and (3) both nodes and relationships may have attributes that are key-value pairs. The name of the relationship is reflected in bold font over the edges. All the nodes with the same label form a group and this leads to an improvement in the query efficiency because a query involving labels limits the

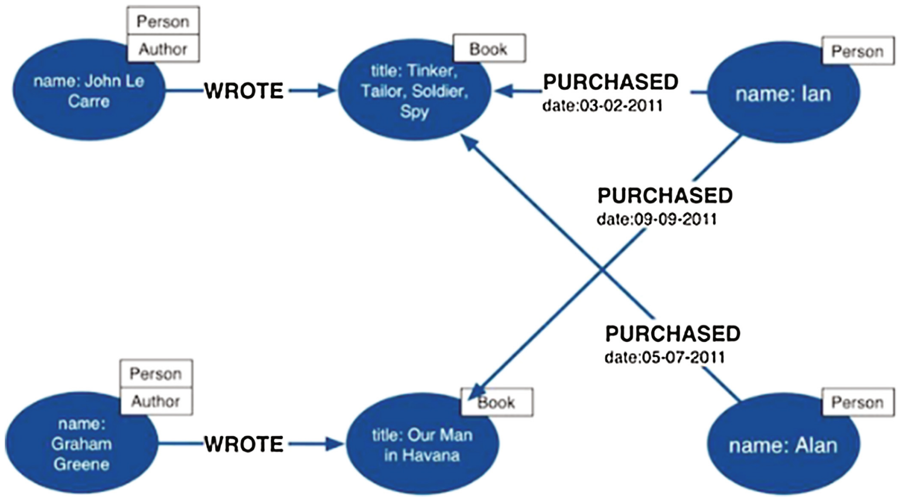


Fig. 2. An example of a property graph [24]

Table 1. Relational schema excerpt. Top *Employee* and Bottom *Organization*

Name	SkypeID	Gender	DOB	employeeID	address	ssn	orgId
Marissa White	marissa@yahoo.com	Female	04-09-1983	mwhite	2600 clifton	1234567890	uc_org
Jason Doe	jason@yahoo.com	Male	04-09-1973	jdoe	3200 clifton	6789083455	uc_org

Name	Location	orgId
University of Cincinnati	2600 Clifton	uc_org

search space to the group of nodes or relationships defined by that label instead of searching through the complete graph [24,52]. As an example, consider the label *Book* in Fig. 2. In this case, when a query is specified to list all books, then only the nodes labelled as *Book* are traversed.

This concludes our brief discussion of the concepts that form the foundation of our research. In the next section, we present our approach and proof-of-concept for transforming relational and RDF databases into a property graph model.

3 Transformation Approach

We leverage the graph transformations proposed by Bouhali et al. [51]. Bouhali et al. focus on converting RDF data into a graph model whereas we envision an extensible approach that embraces model diversity by allowing multiple models such as relational, RDF, and column-family stores from NoSQL databases, for example, all under one framework. Two issues arise: (1) transformation rules to map native model concepts to the property graph model, and (2) assurance that the individual concepts of native models do not get lost even after all the models are transformed into a graph representation. One of the unique characteristics of our proposed model is its native concept-preserving characteristic. This native concept-preserving characteristic is instrumental in facilitating reverse engineering when the graph representation would need to be expressed in the original model terminology. For example, to publish the data for use in a linked data project, RDF would be the model of choice. Thus, data that is represented using any other format would need to be transformed to the RDF model. Furthermore, in the scenario where the integrated, transformed graph includes information from multiple models, having knowledge about which nodes are originating from a particular model offers an independent view of the data models in use. This lays a foundation for model-specific data extraction or transformation.

We now present transformation rules and proof-of-concept of our graph model representation by considering schema excerpts for relational and RDF schemas. For convenience, we use the general term *schema* throughout the paper to refer to both schema and data when discussing models that have a close coupling of the two. For example, mapping a data source to a Neo4j graph database involves mapping structure and instances, but we refer to this activity as *schema mapping*. Similarly for merging two graph databases, we refer to this process as *schema integration*. We intend for the meaning to be clear from the context provided by the discussion and examples.

3.1 Relational Model to a Property Graph

We begin by addressing the conversion of a relational schema into a property graph representation. Das et al. [47] have proposed a methodology for converting relational databases to RDF. Given that Bouhali et al. [51] have proposed an algorithm for transforming RDF to graph, it would appear that we can combine these two proposals [47,51] to transform a relational database to a property graph. However, we do not follow this approach in our work for the following two reasons. First, the final property graph model obtained by Buohali et al. [51] does not reflect the native model features. In their work, they focus on transforming RDF to a graph model and thus graph nodes implicitly correspond to the RDF schema. Our work applies to multiple models, not only RDF, and in anticipation of the need for reverse engineering from property graphs to native models, it is important to preserve the identity of native models in the graph representation. Second, developing transformations from the relational model to the property graph model offers a direct route to the target format (property graph in our case) instead of creating an RDF representation as an intermediate step. Table 1 represents a sample relational schema that we consider for illustrative purposes. The schema excerpt describes two entities, *employee* and *organization*, and the relationship between them using a referential integrity constraint on the attribute *orgId*. We present three transformation rules as follows.

Rule 1: Every tuple in the relation is transformed to a node in the property graph. The node is labelled *RelationalResource* and defines a property in the form of name-value pair as type: *Name of the Relation*. The label serves to disambiguate the relational source from the other models that would also be transformed into a graph representation.

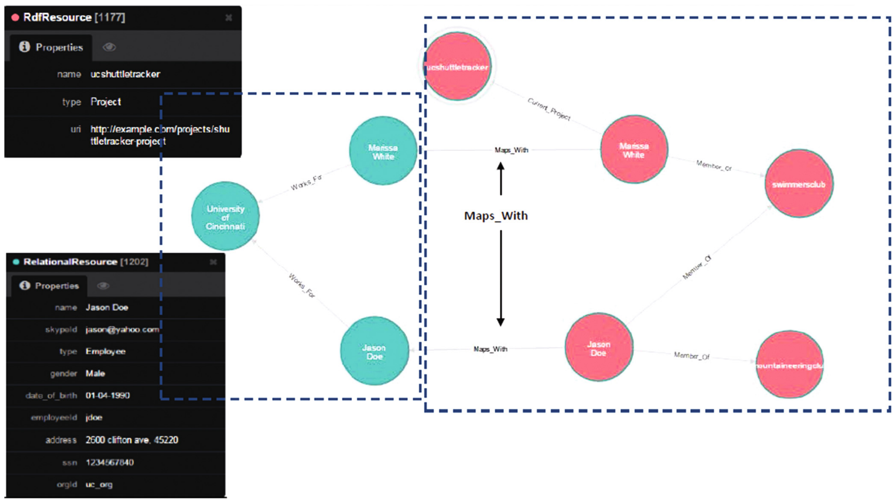
Rule 2: For each of the attributes in the relations, a property (name/value pair) is added to the corresponding node in the graph. This node would be the one that has the value for the property type equal to the name of the relation.

Rule 3: For each foreign key, a relationship is created between the nodes corresponding to the two participating relations.

The dashed rectangle on the left in Fig. 3 illustrate the relational schema from Table 1 as a graph model in Neo4j based on the transformation rules above. The black box at the bottom left shows the properties (name-value pairs) for the employee, “Jason Doe” from the relational schema.

3.2 RDF Model to a Property Graph

We now focus on the RDF schema. Figure 4 presents RDF data based on a schema excerpt from FOAF (Friend Of A Friend) [27] RDF vocabulary. An RDF vocabulary is an RDF schema formed of specific set of classes and properties that


RdfResource [1177]

Properties

name	ucshuttletracker
type	Project
uri	http://example.com/projects/shuttletracker-project

RelationalResource [1202]

Properties

name	Jason Doe
skypeId	jason@yahoo.com
type	Employee
gender	Male
date_of_birth	01-04-1990
employeeId	jdoe
address	2600 clifton ave. 45220
ssn	1234567840
orgId	uc_org

Fig. 3. Excerpts of two heterogeneous schemas originally in different models (left: relational schema, right: RDF schema) unified under a common graph representation

```
<foaf:Group>
 <foaf:name>Swimmersclub</foaf:name>
 <foaf:member>
  <foaf:Person>
         <foaf:name>Marissa White</foaf:name>
         <foaf:homepage rdf:resource="http://homepages.uc.edu/" />
         <foaf:birthday>04-01-1983</foaf:birthday>
         <foaf:gender>Female</foaf:gender>
         <foaf:skypeId>marissa@yahoo.com</foaf:gender>
         <foaf:currentProject rdf:resource="http://www.example.com/projects/ucshuttletracker"/>
  </foaf:Person>
 </foaf:member>
</foaf:Group>
```

Fig. 4. RDF schema excerpt

define resources of a particular domain. The example in Fig. 4 describes two entities *(Person* and *Group)*, the classes that are used to describe them (*foaf:Person* and *foaf:Group*) and the properties that relate them (*foaf:member*). The properties such as *foaf:name* and *foaf:homepage* are applied to a *Person* entity and their values are either literals or resources described using URI (Uniform Resource Identifier). We show only one individual's information in Fig. 4 to save space, but the graph in Fig. 3 shows two individual's information (Marissa White and Jason Doe), their group memberships, and their organization. The black box at the top left shows the properties (key-value pairs) corresponding to the node identified by the name *ucshuttletracker* and of type *Project* in the RDF schema.

With the two input schemas transformed to a graph representation, the next natural question to ask is: *What is the additional merit that the common graph representation offers compared to the knowledge that could have been derived from the native model representations*? Fig. 3 shows both the relational schema and RDF schema connected by mappings *(Maps_With)* in a graph model. This provides an insight into the question. Figure 3 highlights how one can obtain more details for an employee if these two separate schemas can be integrated, compared to the information that we originally received from isolated sources. Relating *Employee* and *Person* nodes, we can identify his or her details such as name and gender and also information on the groups that he or she is a member of, or the homepage. Notice that the information on the homepage of a person is only captured by the RDF schema in our example. By unifying them based on common attributes such as *date of birth* or *skypeId*, an application can benefit from incorporating information from both schemas. This additional information may be harnessed by an organization to develop community-outreach programs based on employee outside interests, for example. Graph models represent a solution for depicting a connected environment. We employ the Neo4j Cypher query language to create mappings *(Maps_With)* between the appropriate nodes by comparing values of certain attributes. In Fig. 3, we link the employee and person information coming from relational and RDF schemas, respectively, based on *skypeId.* In a general context, the example helps to illustrate a use case for leveraging a graph-based model towards a common representation scheme for model and schema diversity.

Apart from facilitating an integrated view, another benefit of our approach is that it preserves the native model concepts in the transformed graph model while providing a uniform representation at the same time. By augmenting nodes with the labels (such as *RelationalResource* and *RDFResource*), one can easily identify information that was originally expressed in a particular native model. At the same time, bringing the individual model and schema concepts under an umbrella of common terms (nodes and relationships) facilitates linking and querying them using a single query language.

In a blog post, the Neo4j developer team present an approach to transform a column-oriented data model to a property graph [49]. The goal is to allow loading of data from a Cassandra data store into Neo4j. The mapping between the source (column-oriented) and target (property graph) data models is taken as input from the user and the resulting graph is created by loading the data from Cassandra to a CSV file. Neo4j supports batch creation of a graph from CSV format. Figure 5 shows a sample schema in Cassandra with *p, r,* and *u* as inputs from the user for schema mapping. The label *p* stands for a property, *r* for a relationship, and *u* for specifying unique constraint field. Since our work incorporates schemas originally expressed in multiple heterogeneous models, we can incorporate the Cassandra to Neo4j mapping by labelling the nodes as *Cassandra Tables.* The approach presented in [49] focuses only on the mapping between one set of source and target data models. In the next section, we present the architecture of our approach.

```
CREATE TABLE playlist.artists_by_first_letter:
    first_letter text: {p}
    artist text: {r}
    PRIMARY KEY (first_letter {p}, artist {u})
CREATE TABLE playlist.track_by_id:
    track_id uuid PRIMARY KEY: {u}
    artist text: {r}
    genre text: {p}
    music_file text: {p}
    track text: {p}
    track_length_in_seconds int: {p}
```

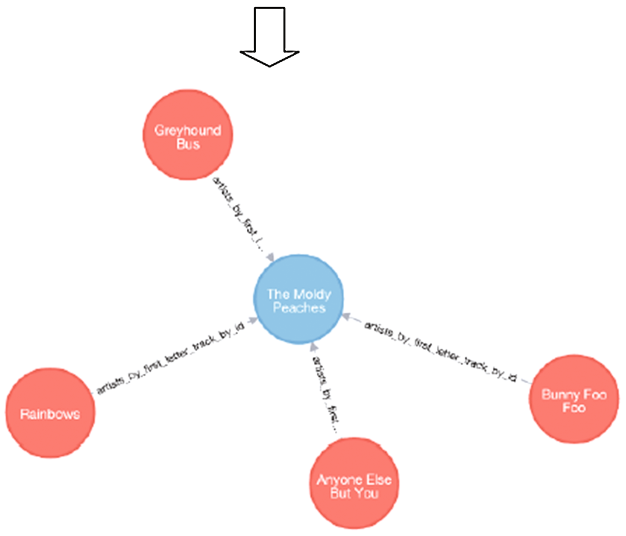


Fig. 5. Placeholders *p,r,* and *u* for schema mapping between a column-oriented store and a property graph model [49]

4 Architecture

Our framework for transforming schemas to a graph-based format can be broken down into three main modules:

1. *The database module* holds schemas and exports a database to CSV format to support the automation step in the application module.
2. *The application module* offers a presentation layer where a user can select the schema that needs to be transformed to Neo4j, and to allow transformation in a systematic manner. The software implementation in this module employs our transformation rules.
3. *The graph module* uses the Neo4j browser to view the transformed schemas.

Figure 6 presents the architecture of our approach.

We use MySQL as the backend database for relational schemas. The process starts at the database layer which consists of three components: schemas,

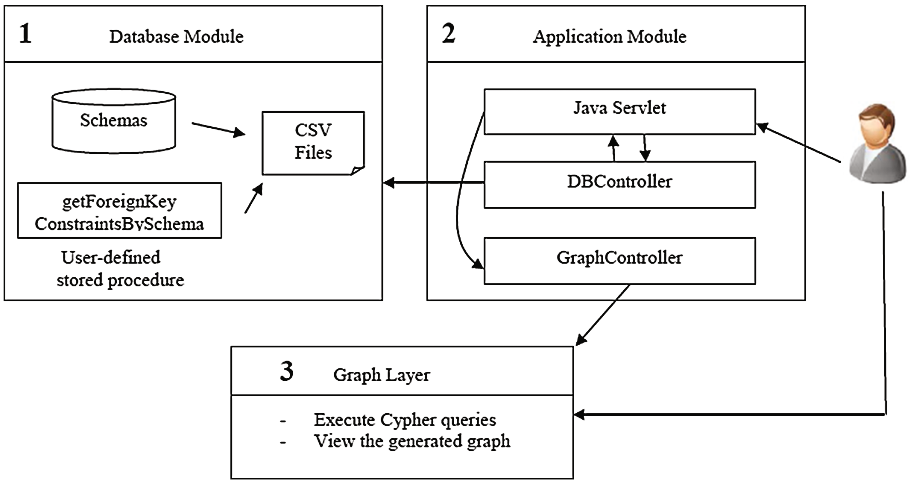


Fig. 6. Architecture of our proposed approach

user-defined stored procedures and the MySQL native export tools. Schemas capture a built-in MySQL database *(information_schema)* and any user defined relational schemas. These user defined schemas are the artifacts that will be transformed to a graph model. The stored procedure reads metadata information from the *information_schema* and identifies all the foreign key relationships in our schema of interest. Figure 7 illustrates a query in our stored procedure to capture all referential integrity constraints. The reason for collecting all the foreign keys in our database is that we need them to create relationships in our graph model based on the transformation rules from Sect. 3.

The user first exports the data in each of the relations in the database as a CSV file using MySQL native export data tool. We use the CSV file format since both the Neo4j community and our programming interface which uses Java support CSV files. Furthermore, Neo4j allows batch creation of a graph from CSV format.

```
select
    concat(table_name, '.', column_name) as 'foreign key',
    concat(referenced_table_name, '.', referenced_column_name) as 'references',
    constraint_name as 'constraint name'
from information_schema.key_column_usage
where referenced_table_name is not null
    and table_schema = 'sakila'
```

Fig. 7. MySQL query to capture foreign key relationships in the MySQL sakila database [26]

```
Class.forName("org.neo4j.jdbc.Driver");
Connection con = DriverManager.getConnection("jdbc:neo4j://localhost:7474/");
Statement stmt = con.createStatement();
ResultSet nodes = stmt.executeQuery("LOAD CSV WITH HEADERS FROM \
"file:C:/Users/usplib/ "+tablename+".csv\" AS line
MERGE (m:RelationalResource {id: line.id, type:'"+tablename+"'})
ON CREATE SET m+= line");
```

Fig. 8. Code-snippet illustrating creation of a Neo4j graph for a relational schema programmatically

At the application level we use Java to interact with the database and the graph modules programmatically. A database controller (DBController) manages connection to the database module and a graph management controller (GraphController) handles connection to Neo4j and submits queries to the graph interface through Java. These three application-level components working together along with our transformation rules from Sect. 3 facilitate automated transformation of a relational schema into a Neo4j property graph.

Figure 8 presents a code snippet that reads a relational schema exported to CSV format and converts it to Neo4j graph. The code focuses on generating nodes which represent the concept of relations in a relational database. Each relation in the schema is exported to a CSV file with the same name as the relation itself. The code reads each of those CSV files and generates nodes with

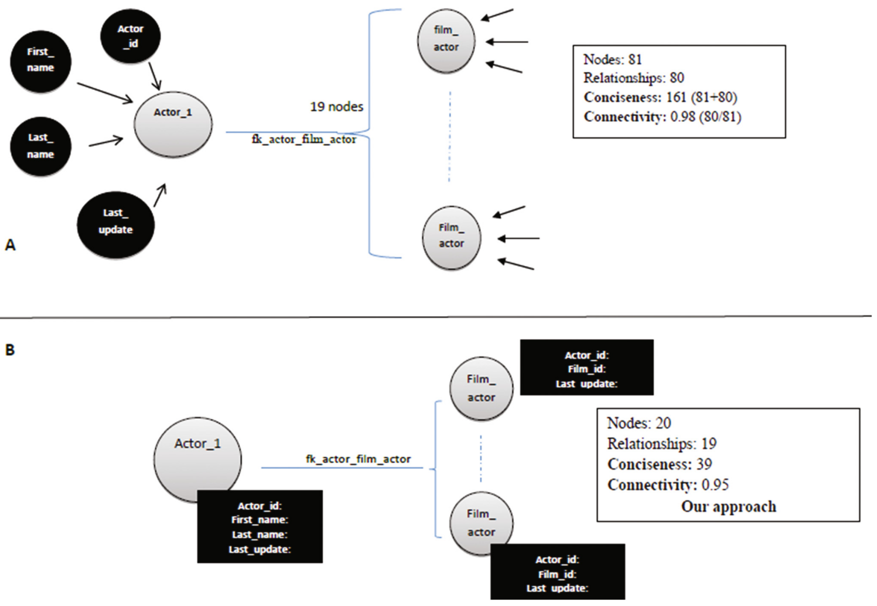


Fig. 9. (a) (top) Modeling attributes as nodes, (b) (bottom) Our approach to modeling attributes as key-value pairs

label *RelationalResource*. The Create Set clause in Fig. 8 adds properties to those nodes. Each field in the CSV file represents one property.

We now have an understanding of the architecture and the input artifacts that are required for software implementation. Our approach is extensible and only requires transformation rules to be defined between any additional models and the property graph.

5 Evaluation

In this section, we discuss both the qualitative and quantitative analysis of our mapping approach. We leverage the evaluation metrics proposed by Buohali et al. [51] and also discuss qualitative merits of our proposal.

The quantitative evaluation metrics we consider are - *conciseness* and *connectivity* of the graph. Conciseness is given by the total number of nodes and relationships and can be used to calculate the graph size. Connectivity is calculated by dividing the number of relationships with the total number of nodes. We apply these measures on the generated graph for an open source MySQL database, sakila, in Table 2.

Table 2. Evaluation metrics results for MySQL database - sakila [26]

Total nodes	47273	Conciseness	62682
Total relationships	15409	Connectivity	0.32

Buohali et al. [51] state that for efficient processing over a graph, connectivity should be at least 1.5, which would signify strong connections in the graph. The connectivity value for our graph is quite low from their benchmark perspective. However, on further investigation, we identify why a low value may not always signify a non-desirable characteristic.

First, according to our transformation rules (Rule 3 in Sect. 3), the only relationship between two nodes that occurs in the target graph model comes from a foreign key relationship in the relational model. This sets the range for the number of relationships between two nodes for a particular constraint to be 0 to max(*n1, n2*) where *n1* and *n2* correspond to the number of each of the two node types. Thus, based on our transformation rules, the number of relationship instances for a particular relationship type (in our case, a foreign key constraint) cannot exceed the number of nodes and hence the connectivity cannot exceed 1. The reason we have an even lower number is that some relations such as *film_text* are not even linked to other relations in the schema.

From this investigation, we come to the conclusion that strong connectivity between nodes in a graph certainly is good for processing but it also does not automatically lead to the conclusion that a lower number is not desirable. The two metrics of conciseness and connectivity can also offer some ideas when we need to make a choice among multiple solutions. A graph with high connectivity is good for processing but if it comes at a price of increasing the graph size (less concise), then this would also lead to an increase in the cost of traversal because of increased path lengths. A larger graph size also implies higher storage requirements.

We evaluated this trade-off between conciseness and connectivity using an alternate mapping and the results are shown in Table 2. For the alternate mapping, we considered modeling attributes as nodes instead of properties (key-value pairs). This resulted in an increase in the number of nodes as well as relationships. The additional relationships come from the new edges created between attributes and entity nodes. We take a small example from sakila database to illustrate the two different mappings and their impact on conciseness and connectivity. Figure 8 shows two graph representations corresponding to two different mappings.

Figure 9a shows a property graph model where attributes are modelled as individual nodes. The conciseness of the graph (which is captured by total number of nodes and relationships) is 161 and connectivity equals 0.98. Our approach shown in Fig. 9b shows how conciseness is increased based on our proposed set of mappings which model attributes as node properties and not as separate nodes. The connectivity does not show much difference and this is because of the nature of the native model and the types of relationships it exhibits. Edges in the graph are generated by foreign key constraints in the relational model.

Figure 9b represents the graph model based on our transformation rules from Sect. 3; Fig. 9a captures an alternate mapping where emphasis is placed on increasing the graph connectivity. We modeled the attributes of a relation as separate nodes and created additional relationships between each of those attributes *(actor_id, first_name, last_name, and last_update)* and the relation node *(actor_1)*. The node labelled *actor_1* represents the data tuple from the actor relation that has *actor_id* equal to one. Similarly, the node *film_actor* represents the relation *film_actor* in the sakila database. The actor node has foreign key relationships with 19 *film_actor* nodes. Based on the *sakila* database, this signifies that the particular actor has acted in 19 films.

Table 3 captures the evaluation metrics from both approaches.

Table 3. Evaluation metrics results for MySQL database - sakila using two mapping transformations

Evaluation criteria	Our approach	Alternate mapping (Fig. 9a)
Total nodes	47273	62967
Total relationships	15409	66239
Conciseness	62682	129206
Connectivity	0.32	1.05

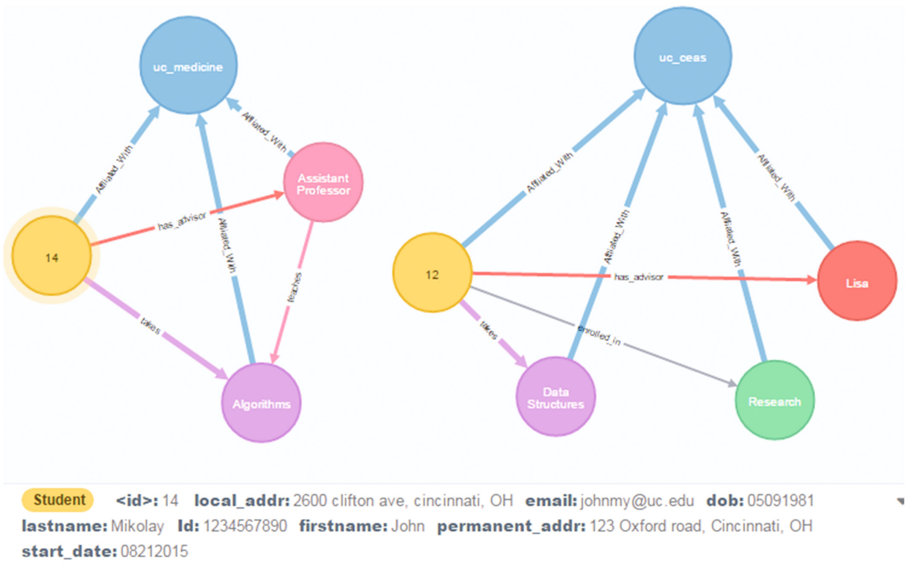
The results from Table 3 and the example from Fig. 9 illustrate two key ideas: (1) the connectivity depends on the nature of original model, and (2) a higher connectivity may come at the cost of an increase in the graph size. Qualitatively, the merit of our proposal for schema mapping lies in the integration between multiple, heterogeneous models under a common graph framework. The *Maps_With* relationship as shown in Fig. 3 creates many additional relationships which were not even present when the transformed graph models of relational and RDF schema were studied separately.

6 Schema Integration over Property Graphs

In this section we propose a framework for schema integration over property graphs. We consider two input schemas expressed as a property graphs and define an algorithm to integrate the two graphs to generate a final integrated schema. The foundation for mapping heterogeneous models to a property graph is established in Sect. 4.

There have been significant research and industry implementations that offer solutions towards schema integration [11–16,23,54]. Solutions exist in the areas of specifying or semi-automatically identifying schema mappings which serve as a foundational step in schema integration. The contribution of our proposal lies in providing an infrastructure that leverages existing mapping algorithms but over a new modelling paradigm, the property graph.

We discuss our schema integration approach using two property graphs shown in Fig. 10. The schemas in both the graphs capture information about entities *student*, *faculty*, *courses*, and *department* and their relationships.



Student <id>: 14 local_addr: 2600 clifton ave, cincinnati, OH email: johnmy@uc.edu dob: 05091981 lastname: Mikolay Id: 1234567890 firstname: John permanent_addr: 123 Oxford road, Cincinnati, OH start_date: 08212015

Fig. 10. Two schemas (left: Medicine Graph, right: Engineering Graph) modelled using property graphs

The two schemas correspond to two different departments: *uc_ceas* and *uc_medicine* shown as the topmost node in Fig. 10.

While there are several commonalities as a result of the common domain, the schemas also exhibit differences creating challenges in schema integration. The differences and our proposed solutions for addressing each of them are as follows:

1. Relationship semantics: The property graph *uc_medicine* allows for a student to have one advisor, if he or she has one (it is optional) while the graph *uc_ceas* requires a student to have an advisor. A student may not be under a research program and thus may not even have an advisor according to the *uc_medicine* schema. We model these min/max constraints using properties *(participation* and *cardinality)* over relationships in our property graphs. As a solution to address this scenario, in the final integrated schema, we impose a min constraint of 0 and max constraint of 1 for the *has_advisor* relationship. The idea behind picking this set of participation and cardinality constraints is that it leads to information preservation of the two native schemas. Following the *uc_medicine* schema in Fig. 10, if a student does not have any advisor, our integrated schema would allow that while also allowing a student to have an advisor if that happens to be the case in the sample schemas.
2. Non-overlapping attributes in two similar entities across the two schemas: this addresses the scenario where an entity in one schema has certain attributes which are not present in the similar entity in the other schema. As an example, the *Faculty Member* node in the *uc_ceas* schema does not store information about faculty's *start_date* and *department* while the *uc_medicine* schema does. This scenario can even be extended to entities and relationships such that one schema may be capturing additional information about a domain that is not covered in the other schema. As an example, consider the entity *ResearchCredits* in the *uc_ceas*. It is not present in the schema for college of medicine. Our approach adds each of the unique attributes from each of the two schemas to the final integrated schema.
3. Differences in constraints: this difference may occur where the properties from two similar entities in the two schemas have different data type or uniqueness constraints [48]. A uniqueness constraint on a property ensures that no two nodes in the graph hold the same value for that property. As an example of the differences, *studentId* for a student entity in *uc_ceas* allows string values while the data type for the corresponding property *Id* in the *uc_medicine* only allows integers. Apart from the difference in data types, the uniqueness constraint for one schema may be composite (consisting of multiple properties) while the other schema may be defining a single-attribute constraint.
   There are multiple ways to resolve this scenario. As an example, if the difference is in terms of data types, then the data type with a wider range of values (String over integer) can be considered for the final integrated schema. However, consider another scenario where the difference is also in terms of number of attributes representing the uniqueness constraint. One schema may have the constraint defined on a set of attributes (composite) while the other

schema uses a single attribute constraint. Considering the possibilities of multiple ways in which differences in the constraints can manifest, we consider the idea of adopting a surrogate key in the final merged schema. The original constraints on the attributes from each of the two schemas will also be copied to the merged schema to prevent any information loss.

4. Difference in field names/entities: entities or attributes across the two schemas may pertain to the same concept but use different names and terminologies. In the example property graphs, the terms *lname* and *lastname* for *Student* node use different terms for the same concept.
   The literature offers numerous solutions for identifying and resolving such conflicts using lexicon, ontology, or string algorithms [17–20].

The final integrated schema addressing the four heterogeneity scenarios above is obtained using the algorithm described in the next section.

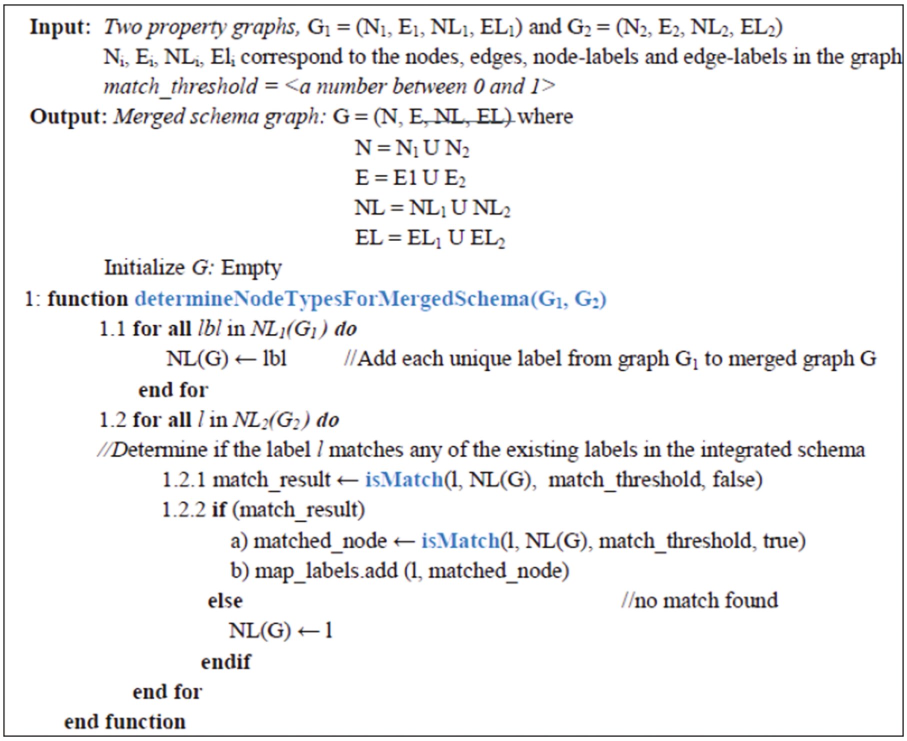
7 Algorithm

The core algorithm can be summarized as follows. Start with one input graph as the base graph. Merge the second into the graph based on a likelihood match for each new node against the base graph. The match between two nodes is determined based on the four solutions described in Sect. 7, algorithms from the literature and a user-defined threshold value. If there is no good match, the node is not merged but added as a new node to the integrated graph. Figures 11, 12 and 13 presents our algorithm as three main modules - *determineNodeTypes-ForMergedSchema, mergeNodes, and mergeRelationships.*

The input schemas modelled using property graphs (G_1 and G_2) are represented using a four-element tuple. The four elements are sets of nodes *(N)*, edges *(E)*, node-labels *(NL)*, and edge-labels *(EL)* in the graph. The notation *NL(G)* and $NL_1(G_1)$ refers to node-labels in the property graph G and G_1 respectively. The output, merged schema is represented as *(N, E, NL, EL)* where each of the set elements is the union of the corresponding elements from the input graphs. The set of nodes N in the merged schema is the union of $N_1(G_1)$ and $N_2(G_2)$. The merged schema is initialized as empty.

The algorithm consists of three main parts: (a) capturing all unique node types from all the input schemas into the final integrated schema, (b) union of nodes N_1 and N_2, and (c) union of relationships E_1 and E_2.

The algorithm proceeds by first identifying the unique node-types across all the input schemas. Each node-type can be considered as an artifact/entity holding a certain set of properties modelled using key-value pairs. Figure 11 shows this module. In our example for schema integration here, we use node labels to capture the node type. The module (Fig. 11) copies all the labels from the first input schema to the integrated schema (Step 1.1). Step 1.2 then iterates over each of the labels in the second schema, G_2 to compare it against all the labels in the merged schema so far. If a match is found that meets a threshold value, then the node is merged with the matched node. The mapping between



```
Input: Two property graphs, $G_1 = (N_1, E_1, NL_1, EL_1)$ and $G_2 = (N_2, E_2, NL_2, EL_2)$
        $N_i$, $E_i$, $NL_i$, $El_i$ correspond to the nodes, edges, node-labels and edge-labels in the graph
        match_threshold = <a number between 0 and 1>
Output: Merged schema graph: G = (N, ~~E, NL, EL)~~ where
                                  $N = N_1 \cup N_2$
                                  $E = E1 \cup E_2$
                                  $NL = NL_1 \cup NL_2$
                                  $EL = EL_1 \cup EL_2$
        Initialize G: Empty
1: function determineNodeTypesForMergedSchema($G_1$, $G_2$)
    1.1 for all lbl in $NL_1(G_1)$ do
          NL(G) ← lbl        //Add each unique label from graph $G_1$ to merged graph G
      end for
    1.2 for all l in $NL_2(G_2)$ do
    //Determine if the label l matches any of the existing labels in the integrated schema
          1.2.1 match_result ← isMatch(l, NL(G),  match_threshold, false)
          1.2.2 if (match_result)
                  a) matched_node ← isMatch(l, NL(G), match_threshold, true)
                  b) map_labels.add (l, matched_node)
                else                                       //no match found
                  NL(G) ← l
                endif
      end for
  end function
```

Fig. 11. Module for unifying the node-types from input schemas

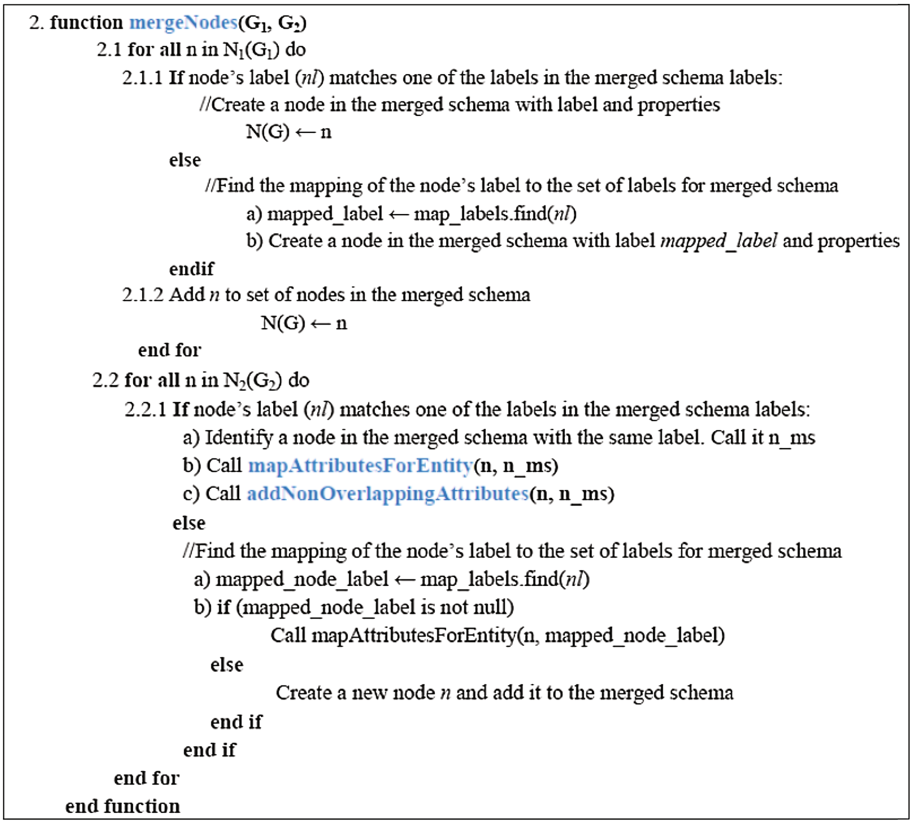
the matched node from the partial merged schema and the new node is also stored (Step 1.2.2).

The algorithm uses the *isMatch* function which takes four arguments: (a) the node, *l*, to be searched, (b) set of node labels *NL(G)* in the merged schema, and (c) a threshold value in the range 0 and 1 for matching, and (d) a boolean argument to signify the type of return value. If the fourth argument is true (Step 1.2.1), it returns the matching node, and if false, a boolean value is returned. This return boolean value signifies if the label l exists in the set of partial merged schema labels or not. If no match is found indicating a new node-type, then the label is added to the set of labels of merged schema (else block in step 1.2.2). The notation *NL(G)* represents set of node-labels for merged schema graph *G*. However, if the node label already exists, then the mapping is stored in a data structure *map_labels* (Step 1.2.2). This function *isMatch* can be customized by applying different schema matching algorithms from the literature.

To understand the rationale behind storing this mapping, refer to our sample input schemas in Fig. 10. We have nodes (*Assistant Professor* and *Lisa*) labelled *Faculty* and *Faculty member*. The labels are representing the same entity but using different terms. The final integrated schema consolidates them into a single label *Faculty*. This mapping information is now important to merge nodes of type *Faculty member* in the *uc_medicine* into nodes with type *Faculty* in the merged schema.

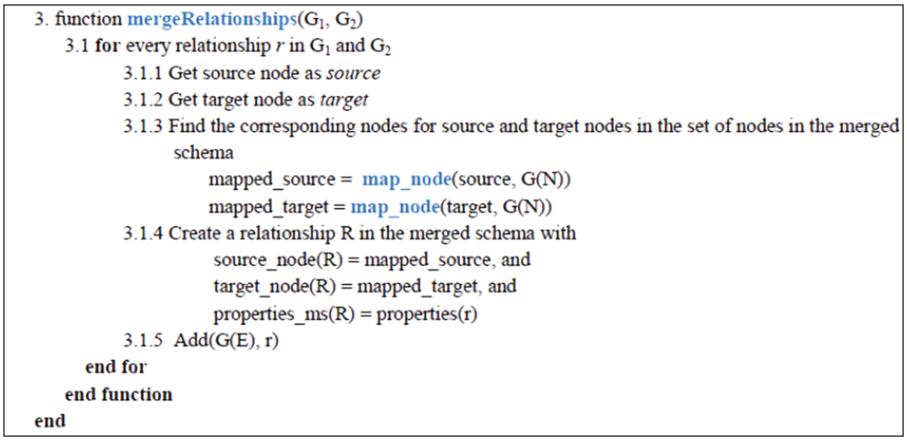
The next steps involve adding nodes and relationships in the final integrated schema. Figures 12 and 13 show the modules addressing this functionality.

After identifying the node-types for the integrated graph, the algorithm iterates through every node in the input graphs. It compares node-type (label) with the set of labels in the integrated schema (Step 2.1.1). If an exact match is not found (for example, *Faculty* and *Faculty Member* node-types in Fig. 10), the closest mapping is found between the current node's label and the set of labels in the integrated schema (step 2.1.1.a in the else block). A new node is then created with the mapped node-type. The notation *N(G)* refers to set of nodes in the merged property graph *G*. Similarly, the algorithm iterates through each node in the next input graph. If the current node's label matches with one of the node's labels in the partial integrated schema, the functions *mapAttributesForEntity* and *addNonOverlappingAttributes* are invoked.



```
2. function mergeNodes($G_1$, $G_2$)
        2.1 for all n in $N_1(G_1)$ do
            2.1.1 If node's label (nl) matches one of the labels in the merged schema labels:
                    //Create a node in the merged schema with label and properties
                        $N(G) \leftarrow n$
                else
                    //Find the mapping of the node's label to the set of labels for merged schema
                        a) mapped_label ← map_labels.find(nl)
                        b) Create a node in the merged schema with label mapped_label and properties
                endif
            2.1.2 Add n to set of nodes in the merged schema
                        $N(G) \leftarrow n$
          end for
        2.2 for all n in $N_2(G_2)$ do
            2.2.1 If node's label (nl) matches one of the labels in the merged schema labels:
                    a) Identify a node in the merged schema with the same label. Call it n_ms
                    b) Call mapAttributesForEntity(n, n_ms)
                    c) Call addNonOverlappingAttributes(n, n_ms)
                else
                    //Find the mapping of the node's label to the set of labels for merged schema
                      a) mapped_node_label ← map_labels.find(nl)
                      b) if (mapped_node_label is not null)
                            Call mapAttributesForEntity(n, mapped_node_label)
                         else
                            Create a new node n and add it to the merged schema
                         end if
                    end if
        end for
    end function
```

Fig. 12. Module for merging the nodes from input schemas



```
3. function mergeRelationships(G_1, G_2)
    3.1 for every relationship r in G_1 and G_2
        3.1.1 Get source node as source
        3.1.2 Get target node as target
        3.1.3 Find the corresponding nodes for source and target nodes in the set of nodes in the merged
              schema
                  mapped_source = map_node(source, G(N))
                  mapped_target = map_node(target, G(N))
        3.1.4 Create a relationship R in the merged schema with
                  source_node(R) = mapped_source, and
                  target_node(R) = mapped_target, and
                  properties_ms(R) = properties(r)
        3.1.5 Add(G(E), r)
      end for
    end function
end
```

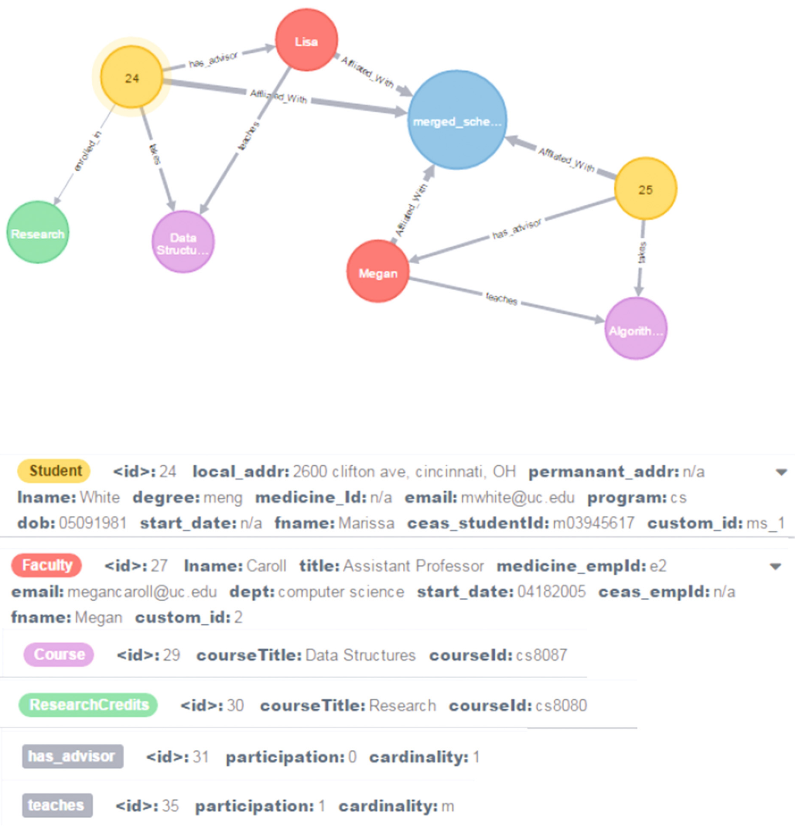
Fig. 13. Module for unifying relationships from input schemas

The final module involves creates edges between nodes in the partial integrated schema. Figure 13 shows this module. The source and target nodes for each relationship in the input graphs is read (Steps 3.1.1 and 3.1.2) and the corresponding nodes in the partial integrated schema are identified (Step 3.1.3). The relationship is then created between the nodes resulting in the final integrated schema (Step 3.1.5).

Figure 14 shows the integrated schema obtained using our algorithm for the sample input property graphs (Fig. 10). Some points to note are as follows.

1. The figure shows the integrated schema for the *Student, Faculty, Course* and *ResearchCredits* node types and one relationship *has_advisor*. Note the surrogate key *custom_id* in the *Student* label. The native primary keys of the individual schemas *uc_ceas* and *uc_medicine* are also preserved. Further, note that the schema *uc_medicine* originally employed the term *FacultyMember* instead of *Faculty*. Using string matching algorithms, we consolidated these two labels into one as *Faculty*.
2. The participation and cardinality constraints for *has_advisor* is 0 and 1, respectively, in the final integrated schema. The original min/max constraints in the native schemas were (0,1) and (1,1), respectively.

The framework provided here merges two property graphs, which can be mapped results from heterogeneous source models. The basic features we address for matching nodes and merging them, for example, can be extended with more sophisticated and powerful techniques from the literature. We provaide a modular proof-of-concept for graph merging.



Student <id>: 24 local_addr: 2600 clifton ave, cincinnati, OH permanant_addr: n/a lname: White degree: meng medicine_Id: n/a email: mwhite@uc.edu program: cs dob: 05091981 start_date: n/a fname: Marissa ceas_studentId: m03945617 custom_id: ms_1

Faculty <id>: 27 lname: Caroll title: Assistant Professor medicine_empId: e2 email: megancaroll@uc.edu dept: computer science start_date: 04182005 ceas_empId: n/a fname: Megan custom_id: 2

Course <id>: 29 courseTitle: Data Structures courseId: cs8087

ResearchCredits <id>: 30 courseTitle: Research courseId: cs8080

has_advisor <id>: 31 participation: 0 cardinality: 1

teaches <id>: 35 participation: 1 cardinality: m

Fig. 14. Integrated schema

8 Evaluation

In this section, we discuss two case studies to evaluate the effectiveness and coverage of our schema integration algorithm [21,22]. The first case study refers to a schema integration example by Petermann et al. [55,56]. The authors provide data models for two heterogeneous systems (enterprise resource planning and customer issue tracking) of a food trading company, and they further employ the models to demonstrate the effectiveness of their graph-based data integration approach [55,56]. We adopt their data sources to test the effectiveness of our approach and compare our integrated schema with their result [55].

For the second case study, we use the schemas modelled by Batini et al. [54]. Through these case study we discuss the features covered by our integrated schema. Our focus is on providing a framework to illustrate schema integration over property graphs that can be further extended and optimized.

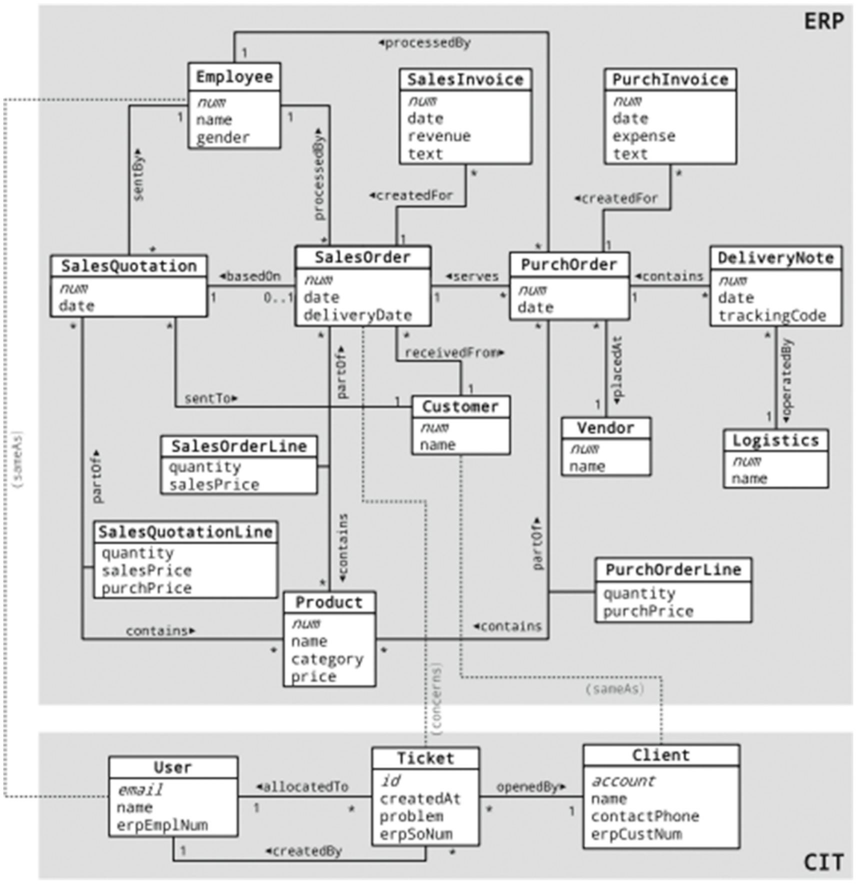
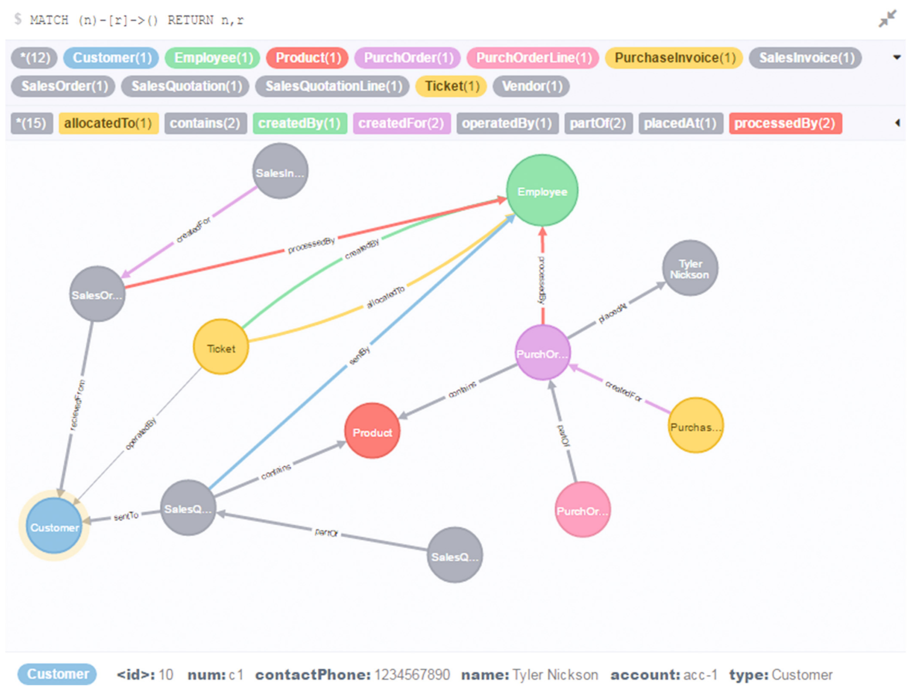


Fig. 15. FoodBroker – Enterprise Resource Planning and Customer Issue Tracking schemas [56]

8.1 Example 1

We consider the *FoodBroker* data source presented by Petermann et al. [56]. The model captures two schemas as shown in Fig. 15. The result obtained through our algorithm (Fig. 16) results in a schema similar to theirs [56]. The *Employee* and *User* nodes are merged along with their attributes into one node *Employee*. Similarly, *Customer* and *Client* entities from the two schemas are consolidated into the *Customer* entity in our integrated schema. In order to highlight schema integration, we constrained sample data in our graph to one instance for each node type. The cardinality constraints (Fig. 15) such as one *SalesInvoice* can be created for multiple instances of *SalesOrder* are captured using properties on the relationships in the graph. Once the interschema relationships are determined, the properties of the similar entities are merged.



```
$ MATCH (n)-[r]->() RETURN n,r
```

Customer <id>: 10 num: c1 contactPhone: 1234567890 name: Tyler Nickson account: acc-1 type: Customer

Fig. 16. Integrated schema obtained through our approach (Example 1)

The added advantage offered by our approach is its ability to handle model diversity. If the native input schemas are in heterogeneous models, the labels of the nodes in their corresponding property graphs (Sect. 3) can be used to capture the data source or model. This allows the schema administrator to preserve native model information while gaining the benefits of collective information as well that is obtained from the integrated schema.

8.2 Example 2

In this example, we consider schema examples modelled by Batini et al. [54]. The input schemas, *Book* and *Publication*, are shown in Fig. 17. The integrated schema obtained through our algorithm is shown in Fig. 18. Our integrated schema creates the same entities (*Publisher, Book, University,* and *Topics*). The main features of our schema are as follows:

1. We also capture the participation and cardinality constraints in our integrated schema (Fig. 18). The *Book* schema (Fig. 16) originally shows a 1:1 relationship between the entities *Book* and *Topics* while *Publication* schema (Fig. 16) exhibits a 1:m relationship between similar entities *Publication* and *Keywords*. In our integrated schema, we resolve this conflict in the cardinalities by modeling the relationship as a 1:m. Batini et al. [54] present and

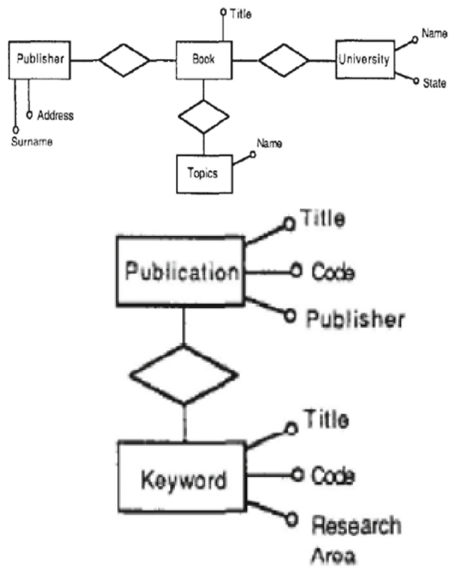
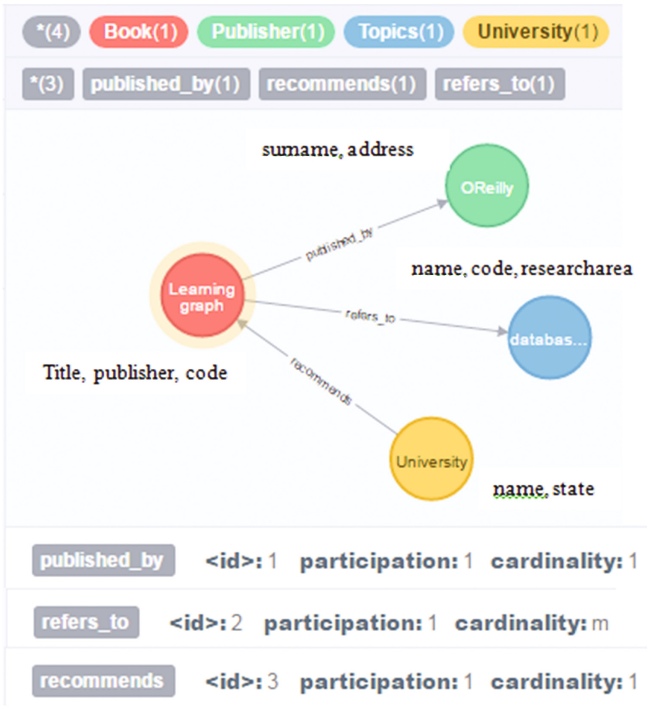


Fig. 17. *Book* and *Publication* schemas [54]



published_by <id>: 1 participation: 1 cardinality: 1

refers_to <id>: 2 participation: 1 cardinality: m

recommends <id>: 3 participation: 1 cardinality: 1

Fig. 18. Integrated schema using our approach (Example 2)

discuss the final integrated schema using a conceptual model (ER). Our approach, based on property graphs, addresses the integration challenge from a graph perspective.
2. We observe that our integrated schema created an additional attribute *Publisher* in the *Book* entity. At this point, our algorithm does not capture the schema conflicts that can arise when some information is modelled as an attribute in one schema and as a relationship in another schema. The information about a book publisher is modelled as a relationship in the *Book* schema and an attribute in the *Publication* schema (Fig. 17). Batini et al. [54] identify and cover this conflict. Our framework can be extended to adress this kind of schema heterogeneity.
3. Batini et al. [54] also show *Book* entity as a subset of the *Publication* entity, considering that publication can also include journals in addition to books. Our work does not yet address identifying and modelling subset relationships. Again, extensions based on the solutions provided in the literature can be incorporated into our framework.

There are numerous opportunities for extending our work to incorporate modules that address a wide variety of heterogeneous features. The main contribution of our work is in providing a framework that employs property graphs as the representation model. Using graph databases allows addressing schema and data integration using one modeling paradigm.

9 Related Work

Data integration and exchange has received significant research attention by both academia and industry practitioners for more than two decades. One of the frequent approaches is defining an intermediate, canonical model that can capture the commonalities and differences of individual, heterogeneous models, thus providing a uniform representation [5–10,55]. Relational, XML, and RDF represent some examples of data models that have been considered for this purpose. We identify two key points that may be raised toward our choice of graph model and discuss each of them below. A discussion of these points highlights the rationale and novelty of our solution.

– *Semantic web technologies facilitate integration by establishing links*

Our idea of employing a property graph model comes from recognizing the data management revolution brought on by big data. In terms of databases, a new class of storage models have emerged called NoSQL databases and graph databases represent one of the categories of the NoSQL family. In this context, we speculate that it would be useful to have frameworks that would allow transformation of different data formats into a model that is amenable to the big data management challenges and our approach represents an effort in this direction. We recognize the immense potential offered by the semantic web research community towards facilitating integration [1,30–32]. RDF model based on subject-object-predicate framework represents the de-facto standard in the linked data and semantic web community and it already leverages a graph model. However, the choice about selecting one model over the other also depends on the problem and domain at hand [29,30,51]. Our vision is to offer an interoperable and integration framework in such a way that it not only facilitates integration of heterogeneous modeling concepts in a flexible and extensible manner, but is also native concept-preserving and aligns with the NoSQL family of data models. Our graph model, as a property graph addresses these requirements.

Furthermore, Bouhali et al. [51] have highlighted potential performance gains in executing large-scale queries over NoSQL graph databases as compared to RDF engines. RDF data needs to be loaded into a SPARQL engine for efficient query performance and authors cite that while contributions have been made towards optimizing query execution in SPARQL but there still remains scope for further improvement in terms of matching up with the performance offered by graph databases for some large-scale queries. Vasilyeva et al. [29] have also compared a graph model with an RDF store and they conclude that since RDF stores both data and metadata using the same format, it requires RDFS and OWL to distinguish the schema from the actual data.

– *What is the advantage of graph based approach over dominant and successful models such as the relational model?*

In comparison to relational databases, data modeling using graphs offers performance gains in processing interconnected data. This is achieved by avoiding the join operation required in the relational model [2,25]. Furthermore, the relational model requires a schema to be defined whereas graph models are flexible [3]. Our work towards schema mapping (Sect. 3) closely aligns with Buohali et al. [51]. Buohali et al. [51] consider translation from RDF to property graphs only. Our application has a broader scope. We build upon their work and extend the scope by making the approach flexible to allow incorporation of additional models.

In terms of schema integration over graph databases, Petermann et al. [55,56] present a system for graph integration and analytics. The system is based on a property graph model and provides three types of graphs: unified metadata graph (UMG), integrated instance graph (IIG) and business transaction graph (BTG). For generating the metadata graph, their approach extracts the schema of objects to translate it into the property graph model. While our contribution on schema integration is closely related to the focus of their work (generating metadata graph for integration), we also include preservation of native model concepts while handling schema mapping. In case of the need for integration over schemas originally expressed in heterogeneous models, our mapping approach (Sect. 3) supports annotation of nodes with labels that define the native data model of the schema elements.

The motivation for employing graph based approach also comes from the fact that graph databases belong to the family of NoSQL data stores. Thus, our framework and algorithm for schema mapping and integration over property graphs lay a foundation for integration of schema-based and schema-less data stores.

10 Conclusion and Future Work

We advocate the idea of employing graph databases as a means of bridging the gap between schema-based and schema-less data stores. Our initial results for schema mapping present a proof-of-concept by illustrating transformation of relational and RDF schemas as the first step. We believe that our approach lays a foundation for addressing the variety aspect of big data and bringing traditional data into a big data environment. The second contribution of our work lies in presenting a schema merging algorithm over property graphs. In this paper, we present a proof-of-concept to illustrate the proposed algorithm. Our approach offers a framework that can be further optimized and it is flexible to incorporate additional schema integration scenarios.

We have translated some traditional models to a property graph. We envision extending our work by incorporating additional data stores. Once we have that achieved that we can incorporate an evaluation study of the transformation process to address the efficiency of the approach. A performance study of querying an integrated graph schema versus disconnected original native schemas is another research direction. The idea of reverse engineering the graph model to obtain the schemas in the original models can also be useful [4,51] to leverage tools from the native data environments.

References

1. Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far. In: Semantic Services, Interoperability and Web Applications: Emerging Concepts, pp. 205–227 (2009)
2. Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., Wilkins, D.: A comparison of a graph database and a relational database: a data provenance perspective. In: Proceedings of the 48th Annual Southeast Regional Conference, p. 42. ACM (2010)
3. Miller, J.J.: Graph database applications and concepts with Neo4j. In: Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA, vol. 2324 (2013)
4. Ruiz, D.S., Morales, S.F., Molina, J.G.: Inferring versioned schemas from NoSQL databases and its applications. In: Conceptual Modeling, pp. 467–480 (2015)
5. Fillottrani, P., Keet, C.M.: Conceptual model interoperability: a metamodel-driven approach. In: Rules on the Web. From Theory to Applications, pp. 52–66. Springer (2014)
6. Bowers, S., Delcambre, L.: On modeling conformance for flexible transformation over data models. In: Proceedings of the ECAI Workshop on Knowledge Transformation for the Semantic Web, pp. 19–26 (2002)
7. Atzeni, P., Cappellari, P., Bernstein, P.A.: Modelgen: model independent schema translation. In: Data Engineering, ICDE, pp. 1111–1112. IEEE (2005)
8. Bernstein, P.A.: Applying model management to classical meta data problems. In: CIDR, pp. 209–220. Citeseer (2003)
9. Atzeni, P., Torlone, R.: MDM: a multiple-data model tool for the management of heterogeneous database schemes. ACM SIGMOD Rec. **26**(2), 528–531 (1997). ACM

10. Bowers, S., Delcambre, L.:, The uni-level description: a uniform framework for representing information in multiple data models. In: Conceptual Modeling-ER 2003, pp. 45–58. Springer (2003)
11. Sheth, A.P. Larson, J.A., Cornelio, A., Navathe, S.B.: A tool for integrating conceptual schemas and user views. In: ICDE, pp. 176–183 (1988)
12. Bellström, P., Kop, C.: Schema quality improving tasks in the schema integration process. Int. J. Adv. Intell. Syst. **7**(3&4), 468–481 (2014). Citeseer
13. Bernstein, P.A., Madhavan, J., Rahm, E.: Generic schema matching, ten years later. Proc. VLDB Endow. **4**(11), 695–701 (2011)
14. Klímek, J., Mlỳnková, I., Nečaskỳ, M.: A framework for XML schema integration via conceptual model. In: International Conference on Web Information Systems Engineering, pp. 84–97. Springer (2010)
15. Bellahsene, Z., Bonifati, A., Rahm, E.: Schema Matching and Mapping, vol. 57. Springer, Heidelberg (2011)
16. Janga, P., Davis, K.C.: Schema extraction and integration of heterogeneous XML document collections. In: International Conference on Model and Data Engineering, pp. 176–187. Springer (2013)
17. Rahm, E.: Towards large-scale schema and ontology matching. In: Schema Matching and Mapping, pp. 3–27. Springer (2011)
18. Cai, Q., Yates, A.: Large-scale semantic parsing via schema matching and lexicon extension, pp. 423–433. Citeseer (2013)
19. Falconer, S.M., Noy, N.F.: Interactive techniques to support ontology matching. In: Schema Matching and Mapping, pp. 29–51. Springer (2011)
20. Cheatham, M., Hitzler, P.: String similarity metrics for ontology alignment. In: International Semantic Web Conference, pp. 294–309. Springer (2013)
21. Doan, A., Halevy, A.Y.: Semantic integration research in the database community: a brief survey. AI Mag. **26**(1), 83 (2005)
22. Vaidyanathan, V.: A Metamodeling Approach to Merging Data Warehouse Conceptual Schemas, University of Cincinnati (2008)
23. Bernstein, P., Ho, H.: Model management and schema mappings: theory and practice. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 1439–1440 (2007). VLDB Endowment
24. Property Graph. http://neo4j.com/developer/graph-database. Accessed 27 Jan 2016
25. Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O'Reilly Media Inc., Sebastopol (2013)
26. Sakila Sample Database. https://dev.mysql.com/doc/sakila/en/. Accessed 14 Mar 2016
27. FOAF Vocabulary Specification 0.99 (2014). http://xmlns.com/foaf/spec/. Accessed 27 Jan 2016
28. Google Trends. https://www.google.com/trends/. Accessed 18 Mar 2016
29. Vasilyeva, E., Thiele, M., Bornhövd, C., Lehner, W.: Leveraging flexible data management with graph databases. In: First International Workshop on Graph Data Management Experiences and Systems (GRADES). ACM (2013). ISBN: 978-1-4503-2188-4, Article 12. http://doi.acm.org/10.1145/2484425.2484437, doi:10.1145/2484425.2484437
30. Goble, C., Stevens, R.: State of the nation in data integration for bioinformatics. J. Biomed. Inf. **41**(5), 687–693 (2008). Elsevier
31. Halevy, A.Y., Ives, Z.G., Mork, P., Tatarinov, I.: Piazza: data management infrastructure for semantic web applications. In: Proceedings of the 12th International Conference on World Wide Web, pp. 556–567. ACM (2003)

32. Halevy, A., Rajaraman, A., Ordille, J.: Data integration: the teenage years. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 9–16 (2006). VLDB Endowment
33. Big Data and Analytics. https://www.idc.com/prodserv/4Pillars/bigdata. Accessed 27 Jan 2016
34. Cloud Platform Storage: Relational vs. Scale-Out. http://davidchappellopinari.blogspot.com/2009/02/cloud-platform-storage-relational-vs.html. Accessed 14 Mar 2016
35. Özcan, F., Tatbul, N., Abadi, D.J., Kornacker, M., Mohan, C., Ramasamy, K., Wiener, J.: Are we experiencing a big data bubble? In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD 2014), Snowbird, Utah, USA, pp. 1407–1408 (2014)
36. Chattopadhyay, B., Lin, L., Liu, W., Mittal, S., Aragonda, P., Lychagina, V., Kwon, Y., Wong, M.: Tenzing a SQL implementation on the MapReduce framework. In: Proceedings of VLDB, pp. 1318–1327 (2011)
37. Teradata Aster Analytics. http://www.teradata.com/Teradata-Aster-SQL-MapReduce. Accessed 27 Jan 2016
38. Sherif, S.: Use SQL-like languages for the MapReduce framework. http://www.ibm.com/developerworks/library/os-mapreducesql/os-mapreducesql-pdf.pdf. Accessed 27 Jan 2016
39. SQL-on-Hadoop, Landscape and Considerations. https://www.mapr.com/why-hadoop/sql-hadoop/sql-hadoop-details. Accessed 27 Jan 2016
40. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive - a petabyte scale data warehouse using Hadoop. In: Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, 1–6 March 2010, Long Beach, California, USA, pp. 996–1005 (2010)
41. Floratou, A., Minhas, U.F., Özcan, F.: SQL-on-Hadoop: full circle back to shared-nothing database architectures. Proc. VLDB Endow. **7**, 1295–1306 (2014)
42. Hitzler, P., Janowicz, K.: Linked data, big data, and the 4th paradigm. Semant. Web **4**(3), 233–235 (2013)
43. Nejdl, W., Wolpers, M., Capelle, C.: The RDF schema specification revisited. In: Workshop Modellierung (2000)
44. RDF Vocabulary Description Language 1.0: RDF Schema (2002). https://www.w3.org/2001/sw/RDFCore/Schema/200203/. Accessed 27 Jan 2016
45. Cypher query language. http://neo4j.com/developer/cypher-query-language/. Accessed 27 Jan 2016
46. Neo4j ranking. http://db-engines.com/en/ranking/graph+dbms. Accessed 27 Jan 2016
47. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF mapping language (W3C recommendation) (2012). https://www.w3.org/TR/r2rml/. Accessed 27 Jan 2016
48. The Neo4j Java Developer Reference v3.0 (2016). http://neo4j.com/docs/java-reference/current/#transactions-unique-nodes
49. Lyon, W.: Neo4j + Cassandra: Transferring Data from a Column Store to a Property Graph (2016). https://neo4j.com/blog/neo4j-cassandra-transfer-data/
50. Hecht, R., Jablonski, S.: NoSQL evaluation: a use case oriented survey. In: International Conference on Cloud and Service Computing (CSC), pp. 336–341 (2011)
51. Bouhali, R., Laurent, A.: Exploiting RDF open data using NoSQL graph databases. In: 11th IFIP WG 12.5 International Conference on Artificial Intelligence Applications and Innovations, AIAI, Bayonne, France, pp. 177–190, 14–17 September 2015

52. Neo4j database. http://neo4j.com/. Accessed 27 Jan 2016
53. Resource Description Framework (RDF) Schema Specification 1.0. https://www.w3.org/TR/2000/CR-rdf-schema-20000327/. Accessed 27 Jan 2016
54. Batini, C., Lenzerini, M., Navathe, S.B.: A comparative analysis of methodologies for database schema integration. ACM Comput. Surv. (CSUR) **18**(4), 323–364 (1986)
55. Petermann, A., Junghanns, M., Mller, R., Rahm, E.: Graph-based data integration and business intelligence with BIIIG. Proc. VLDB Endow. **7**(13), 1577–1580 (2014)
56. Petermann, A., Junghanns, M., Mller, R., Rahm, E.: FoodBroker-generating synthetic datasets for graph-based business analytics. In: Workshop on Big Data Benchmarks, pp. 145–155. Springer (2014)

Modeling Terminologies for Reusability in Faceted Systems

Daniel R. Harris(✉)

Department of Computer Science, Center for Clinical and Translational Science, College of Engineering, University of Kentucky, Lexington, KY, USA
daniel.harris@uky.edu

Abstract. We integrate heterogeneous terminologies into our category-theoretic model of faceted browsing and show that existing terminologies and vocabularies can be reused as facets in a cohesive, interactive system. Commonly found in online search engines and digital libraries, faceted browsing systems depend upon one or more taxonomies which outline the structure and content of the facets available for user interaction. Controlled vocabularies or terminologies are often curated externally and are available as a reusable resource across systems. We demonstrated previously that category theory can abstractly model faceted browsing in a way that supports the development of interfaces capable of reusing and integrating multiple models of faceted browsing. We extend this model by illustrating that terminologies can be reused and integrated as facets across systems with examples from the biomedical domain. Furthermore, we extend our discussion by exploring the requirements and consequences of reusing existing terminologies and demonstrate how categorical operations can create reusable groupings of facets.

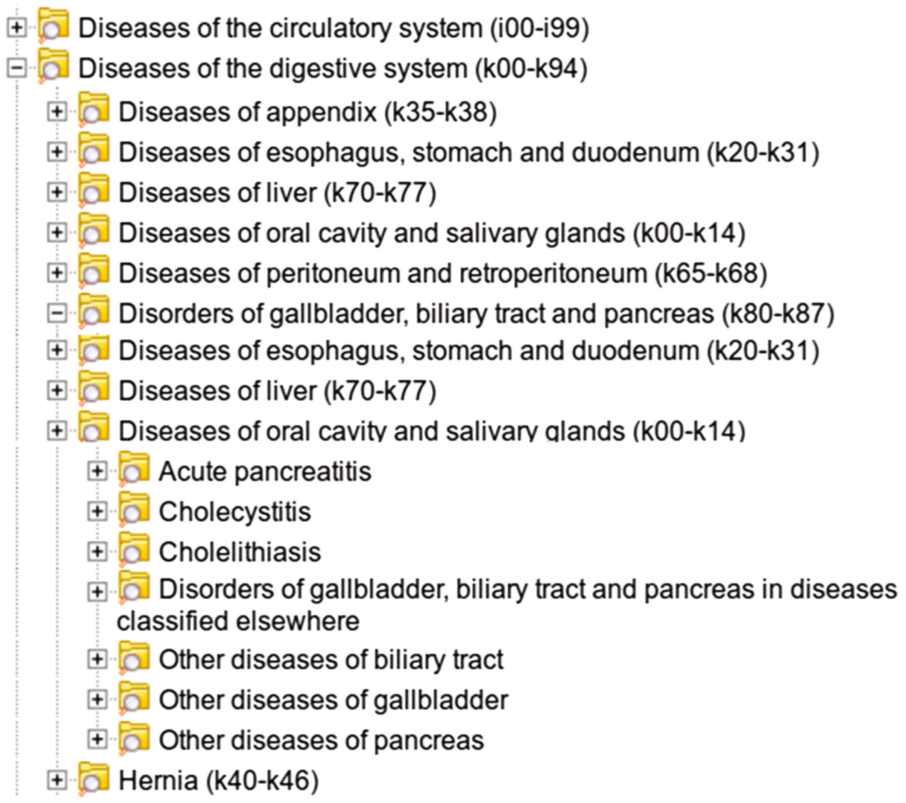
Keywords: Faceted browsing · Terminologies · Category theory · Information reuse

1 Introduction

Faceted classification is the process of assigning facets to resources in a way that enables intelligent exploratory search aided by an interactive faceted taxonomy [30]. Exploratory search using a faceted taxonomy is often called faceted browsing (or faceted navigation or faceted search) [14] and is commonly found in digital libraries or online search engines. Facets are the individual elements of the faceted taxonomy and are simply attributes known to describe an object being cataloged; these collections of facets are often organized as sets, hierarchies, lattices, or graphs. Facets are usually shown alongside a list of other related, relevant facets that aid in interactive filtering and expansion of search results [15]. A simple example of facets for a digital library of books would be genre or publication date. The taxonomy behind the interface is either custom to the search needs of the interface or bootstrapped by a terminology familiar to those

S.H. Rubin and T. Bouabana-Tebibel (eds.), *Quality Software Through Reuse and Integration*, Advances in Intelligent Systems and Computing 561,
DOI 10.1007/978-3-319-56157-8_7

with working knowledge of the domain. In the biomedical domain, patients are often classified according to ICD10 diagnosis codes [31] in their electronic health record; as seen in Fig. 1, the i2b2 query tool is capable of searching for patients using ICD10 codes [19] as well as other common biomedical terminologies. We will discuss i2b2 and another biomedical application in Sect. 4.



- Diseases of the circulatory system (i00-i99)
- Diseases of the digestive system (k00-k94)
  - Diseases of appendix (k35-k38)
  - Diseases of esophagus, stomach and duodenum (k20-k31)
  - Diseases of liver (k70-k77)
  - Diseases of oral cavity and salivary glands (k00-k14)
  - Diseases of peritoneum and retroperitoneum (k65-k68)
  - Disorders of gallbladder, biliary tract and pancreas (k80-k87)
  - Diseases of esophagus, stomach and duodenum (k20-k31)
  - Diseases of liver (k70-k77)
  - Diseases of oral cavity and salivary glands (k00-k14)
    - Acute pancreatitis
    - Cholecystitis
    - Cholelithiasis
    - Disorders of gallbladder, biliary tract and pancreas in diseases classified elsewhere
    - Other diseases of biliary tract
    - Other diseases of gallbladder
    - Other diseases of pancreas
  - Hernia (k40-k46)

Fig. 1. Users can select from a variety of biomedical facets within i2b2, including those from existing and well-known terminologies; a subset of the ICD10 terminology as viewed through the i2b2 query tool is shown here.

Facet models formalize faceted data representations and the interactive operations that follow for exploratory search tasks. Wei et al. observed three major theoretical foundations behind current research of facet models: set theory, formal concept analysis, and lightweight ontologies [30]. In our previous work, we demonstrated that category theory can act as a theoretical foundation for faceted browsing that encourages reuse and interoperability by uniting different facet models together under a common framework [9,10]. We also established facets and faceted taxonomies as categories and have demonstrated how the computational elements of category theory, such as products and functors, extend the utility of our model [9]. The usefulness of faceted browsing systems is well-established in the digital libraries research community [8,20], but reuse and interoperability are typically not major design considerations [9]. Our goal is to create a rich environment for faceted browsing where reuse and interoperability are primary design considerations.

In this extended paper [11], we integrate heterogeneous terminologies as facets into the category-theoretic model of faceted browsing [10] so that existing and well-known terminologies can be reused in an intelligent manner. These terminologies themselves can act as a faceted taxonomy, but we also demonstrate the usefulness of modeling a terminology as a facet type. We discuss how to create instances of facets and faceted taxonomies in order for our model to interact with multiple, heterogeneous sources. In our extension, we show that categorical pushout and pullback operations help construct reusable groupings of facets. We demonstrate how multiple terminologies can coexist, work together efficiently, and contribute toward the ultimate goal of a particular faceted interface. We present and compare two considerations for modeling faceted browsing interfaces that utilize multiple terminologies: the need to merge facets together into a single "master" taxonomy and the need for multiple focuses from different terminologies.

2 Background

We must discuss faceted taxonomies and introduce concepts from category theory before discussing our category-theoretic model of faceted browsing and its extensions.

2.1 Faceted Taxonomies

At the heart of faceted browsing, regardless of the facet model chosen for a particular interface, there lies a taxonomy which organizes and gives structure to the facets that describe the resources to be explored. Faceted taxonomies can aid in the construction of information models or aid in the construction of a larger ontology [4,22]. If facet browsing is truly a pivotal element to modern information retrieval [7], then great care must be taken to abstractly model and fully integrate the taxonomies behind the interface. Depending upon the needs and complexity of its design, a faceted browsing interface may rely upon one or many faceted taxonomies to drive exploration and discovery.

2.2 Category Theory

Category theory has been useful in modeling problems from multiple science domains [25], including physics [6], cognitive science [21], and computational biology [26]. Categories also model databases [23,25] where migration between schemas can be represented elegantly [24]. We will demonstrate that facets and schemas are structurally related in Sect. 3.2.

In this section, we introduce a few concepts from category theory that are necessary for understanding our model. Informally, a category $\mathcal{C}$ is defined by stating a few facts about the proposed category (specifying its objects, morphisms, identities, and compositions) and demonstrating that they obey identity and associativity laws [25].

Definition 1. *A category $\mathcal{C}$ consists of the following:*

1. *A collection of objects, $Ob(\mathcal{C})$.*
2. *A collection of morphisms (also called arrows). For every pair $x, y \in Ob(\mathcal{C})$, there exists a set $Hom_{\mathcal{C}}(x, y)$ that contains morphisms from x to y; a morphism $f \in Hom_{\mathcal{C}}(x, y)$ is of the form $f : x \to y$, where x is the domain and y is the codomain of f.*
3. *For every object $x \in Ob(\mathcal{C})$, the identity morphism, $id_x \in Hom_{\mathcal{C}}(x, x)$, exists.*
4. *For $x, y, z \in Ob(\mathcal{C})$, the composition function is defined as follows: $\circ : Hom_{\mathcal{C}}(y, z) \times Hom_{\mathcal{C}}(x, y) \to Hom_{\mathcal{C}}(x, z)$.*

Given 1-4, the following laws hold:

1. *identity: for every $x, y \in Ob(\mathcal{C})$ and every morphism $f : x \to y$, $f \circ id_x = f$ and $id_y \circ f = f$.*
2. *associativity: if $w, x, y, z \in Ob(\mathcal{C})$ and $f : w \to x$, $g : x \to y$, $h : y \to z$, then $(h \circ g) \circ f = h \circ (g \circ f) \in Hom_{\mathcal{C}}(w, z)$.*

Our model of faceted browsing leverages two well-known categories: **Rel** and **Cat**. We leverage these as building blocks in our model by creating subcategories: categories constructed from other categories by taking only a subset of their objects and the necessary corresponding morphisms.

Definition 2. ***Rel*** *is the category of sets as objects and relations as morphisms [1], where we define relation arrows $f : X \to Y \in Hom_{\boldsymbol{Rel}}(X, Y)$ to be a subset of $X \times Y$.*

Definition 3. ***Cat*** *is the category of categories. The objects of* ***Cat*** *are categories and the morphisms are functors (mappings between categories).*

Functors can informally be thought of as mappings between categories, but additional conditions are required:

Definition 4. *A functor F from category $\mathcal{C}_1$ to $\mathcal{C}_2$ is denoted $F : \mathcal{C}_1 \to \mathcal{C}_2$, where $F : Ob(\mathcal{C}_1) \to Ob(\mathcal{C}_2)$ and for every $x, y \in Ob(\mathcal{C}_1)$, $F : Hom_{\mathcal{C}_1}(x, y) \to Hom_{\mathcal{C}_2}(F(x), F(y))$. Additionally, the following must be preserved:*

1. *identity: for any object $x \in Ob(\mathcal{C}_1)$, $F(id_{\mathcal{C}_1}) = id_{F(\mathcal{C}_1)}$.*
2. *composition: for any $x, y, z \in Ob(\mathcal{C}_1)$ with $f : x \to y$ and $g : y \to z$, then $F(g \circ f) = F(g) \circ F(f)$.*

In this section, we describe our category-theoretic model of faceted browsing. We demonstrated previously that our model encourages and facilitates reuse and interoperability within and across faceted browsing systems; we describe only the key elements and leave the minor details available in our prior work [9].

Definition 5. *Let* $\boldsymbol{Tax}$ *be a sub-category of* $\boldsymbol{Rel}$*, the category of sets as objects and relations as morphisms where* $Ob(\boldsymbol{Tax}) = Ob(\boldsymbol{Rel})$ *and let the morphisms be the relations that correspond only to the* $\subseteq$ *relations. The identity and composition definitions are simply copied from* $\boldsymbol{Rel}$*.*

Tax is simply a slimmer version of **Rel**, where we know exactly what binary relation is being used to order the objects. In our previous work, we did not apply a name to **Tax** and left this category described as **Rel** restricted to inclusion mappings [9]; applying a name allows us to be concise in our discussions, which is important because **Tax** will be the building block that will allow us to apply the additional structure and granularity needed to support faceted browsing. We can refer to an independent facet, such as genre, language, or price-range, as a *facet type.*

Definition 6. *A facet type (a facet* i *and its related sub-facets) of a faceted taxonomy is a sub-category of* $\boldsymbol{Tax}$*, the category of sets as objects and inclusion relations as morphisms. Let us call this sub-category* $\boldsymbol{Facet}_i$ *and let* $Ob(\boldsymbol{Facet}_i) \subseteq Ob(\boldsymbol{Tax})$ *with the morphisms being the corresponding* $\subseteq$ *relations for those objects. The relevant identity and composition definitions are also copied from* $\boldsymbol{Tax}$*.*

From this facet type, users make focused selections when drilling down into faceted data. This selection pinpoints a subset of the facets within this type and by proxy, it pinpoints a subset of the resources classified.

Definition 7. *We can define a subcategory of* $\boldsymbol{Facet}_i$*, called* $\boldsymbol{Focus}_i$*, to represent a focused selection of objects from* $\boldsymbol{Facet}_i$ *having* $Ob(\boldsymbol{Focus}_i) \subseteq Ob(\boldsymbol{Facet}_i)$ *and the necessary corresponding morphisms, identity, and composition definitions for those objects.*

Each individual facet category belongs to a larger taxonomy that collectively represents the structure of information within a facet browsing system.

Definition 8. *Let* $\boldsymbol{FacetTax}$ *be a category that represents a faceted taxonomy, whose objects are the disjoint union of* $\boldsymbol{Facet}_i$ *categories. In other words, let* $Ob(\boldsymbol{FacetTax}) = \coprod_{i=1}^{n} \boldsymbol{Facet}_i$ *and* $n = |Ob(\boldsymbol{FacetTax})|$*. The morphisms of* $\boldsymbol{FacetTax}$ *are functors (mappings between categories) of the form* $Hom_{\boldsymbol{FacetTax}}(\mathcal{C}, \mathcal{D}) = \{F : \mathcal{C} \rightarrow \mathcal{D}\}$*.*

Once you have a faceted taxonomy constructed, interactivity and engagement with it follows; a natural task for users of a faceted system is to perform queries that focus and filter objects being explored.

Definition 9. *A facet universe,* U*, is the* n*-ary product* [1] *within the* $\boldsymbol{FacetTax}$ *category, defined as* $\prod_{i=1}^{n} \boldsymbol{Facet}_i$*, where* $n = |Ob(\boldsymbol{FacetTax})|$*. The* n *coordinates of* U *are projection functors* $P_j : \prod \boldsymbol{Facet}_i \rightarrow \boldsymbol{Facet}_j$*, where* $j = 1, \ldots, n$ *is the* j*th projection of the* n*-ary product.*

Note that since $\mathbf{Focus}_i$ is a subcategory of $\mathbf{Facet}_i$, there exists a restricted universe $U_{\subseteq} \subseteq U$ where every facet is potentially reduced to a focused subset. The act of querying the universe is essentially constructing this restricted universe $U_{\subseteq}$.

Definition 10. *A faceted query, Q, is the modified n-ary product* [1] *within the **FacetTax** category, defined as*
$\prod_{i=1}^{n} \boldsymbol{Focus}_i$, where $n = |Ob(\boldsymbol{FacetTax})|$. The n coordinates of Q are similarly defined as projection functors $P_j : \prod \boldsymbol{Focus}_i \rightarrow \boldsymbol{Focus}_j$.

2.3 A Category-Theoretic Model

We visually summarize the key containers and products in Fig. 2. We will later demonstrate that this same faceted taxonomy can be represented as a graph. The objects of each $\mathbf{Facet}_i$ are sets of resources that have been classified as belonging to that facet type; our model can reuse the facets and adjust the surrounding structure to fit our needs: if we wish to arrange the facets as graphs, we can do so without bothering the resource and facet linkages.

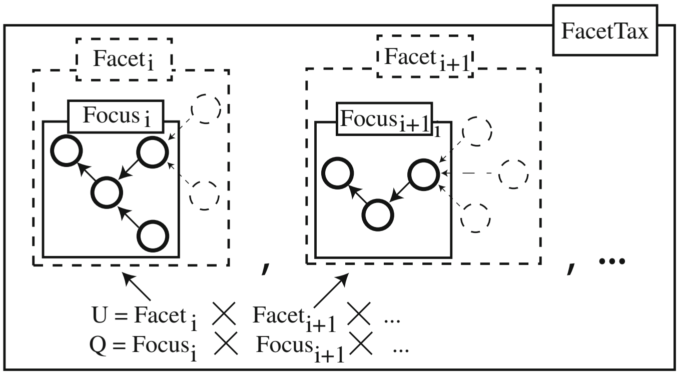


Fig. 2. The structure of facet, focus, and taxonomy are easy to visualize due to their natural hierarchical relationships. Universes and queries are products utilizing this structure.

Figure 3 shows a sample piece of a medication taxonomy; each resource is classified using the taxonomy. In our model, we refer to resources in the general sense. The type of resource depends upon the interface: resources could be books in a digital library system, documents in a electronic health system, and so on. Note that the taxonomy in Fig. 3 could easily be considered the facet type *medications*, which belongs to a large taxonomy (not pictured) instead of a complete faceted taxonomy to itself; either scenario is acceptable as this will depend upon the design of the faceted browsing system, which can vary.

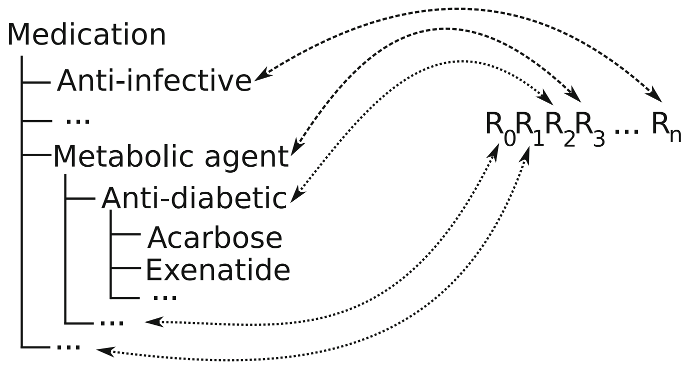


Fig. 3. We show a sample faceted taxonomy for medications. The objects of each **Facet** are pointers to a resource that has been classified as belonging to that particular facet type.

3 Leveraging Multiple Terminologies

The category-theoretic model is perfectly capable of representing basic faceted interfaces in its current form, but the ability to model and interact with multiple heterogeneous sources is needed to support more intricate interfaces. The capacity to integrate multiple terminologies rests largely upon our ability to model *instances* of our facet categories. Understanding the relationship between schemas and facets will be key to understanding the process for creating instances.

In our previous work on modeling faceted browsing for reusability, we demonstrated the importance that graphs play in reusing and integrating models [9]. We confirm this importance in the following sub-sections.

3.1 Underlying Graphs

The ability to transform into other structures enables the category theoretic model of faceted browsing to consume other models. We show that graphs underlie categories and that a graph-based representation of a facet can be used as input in modeling taxonomies.

Definition 11. ***Grph*** *is the category with graphs as objects. A graph G is a sequence where $G := (V, A, src, tgt)$ with the following:*

1. *a set V of vertices of G*
2. *a set A of edges of G*
3. *a source function $src : A \rightarrow V$ that maps arrows to their source vertex*
4. *a target function $tgt : A \rightarrow V$ that maps arrows to their target vertex*

Definition 12. *The graph underlying a category $\mathcal{C}$ is defined as a sequence $U(\mathcal{C})$ = (Ob($\mathcal{C}$), $Hom_{\mathcal{C}}$, dom, cod)* [25].

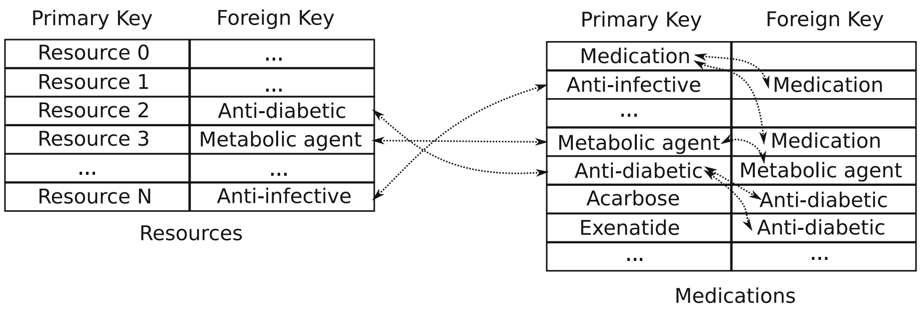
We previously demonstrated given that there exists a functor $U : \mathbf{Cat} \rightarrow \mathbf{Grph}$, so **FacetTax** can produce graphs of $\mathbf{Facet}_i$ categories for $i = (1, \ldots, |Ob(\mathbf{FacetTax})|)$ [9].

Definition 13. *Let $U(\boldsymbol{Facet}_i)$ be the underlying graph of an individual facet and let $U(\boldsymbol{FacetTax})$ be the underlying graph of the faceted taxonomy at large, as constructed and detailed above.*

This underlying graph will be important in discussing the relationship between schemas and faceted taxonomies, which will allow us to create instances of facets and faceted taxomonies.

3.2 Facet and Schema

In this section, we describe how to create instances of facets and faceted taxonomies with a method and rationale that is inspired by Spivak's database schemas [25]. In fact, we discover that facets are equivalent to database schemas. Although this equivalence may be unexpected initially, conceptually the idea of a database schema is not unlike facets when viewed from a category theory perspective: both describe the conceptual layout that organizes information (rows/entities in the case of databases and resources in the case of facets). Figure 4 shows the same faceted information found in Fig. 3, but within a schema. Note that parts of the table are abbreviated with ellipses in order to save space. We will discuss these tables and their relationship with faceted browsing in detail in the next section.



Primary Key	Foreign Key
Resource 0	...
Resource 1	...
Resource 2	Anti-diabetic
Resource 3	Metabolic agent
...	...
Resource N	Anti-infective

Resources

Primary Key	Foreign Key
Medication	
Anti-infective	Medication
...	
Metabolic agent	Medication
Anti-diabetic	Metabolic agent
Acarbose	Anti-diabetic
Exenatide	Anti-diabetic
...	...

Medications

Fig. 4. A resource table and a medications table using example data from Fig. 3 shows the role that primary and foreign keys play in modeling faceted browsing.

Preliminary Definitions. Spivak's definition of schemas depends upon the idea of congruence, which in turn depends on defining paths, path concatenation, and path equivalence declarations [25].

Definition 14. *If $G := (V, A, src, tgt)$ is a graph, then a path of length n in G is a sequence of arrows denoted $p \in Path_G^{(n)}$, where $Path_G$ is the set of paths in G [25].*

Definition 15. *Given a path $p : v \rightarrow w$ and $q : q \rightarrow x$, $p + +q : v \rightarrow x$ is the concatenation of the two paths* [25].

Definition 16. *A path equivalence declaration (abbreviated by Spivak as PED) is an expression of the form $p \simeq q$, where $p, q \in Path_G$ have the same source and target, e.g., $src(p) = src(q)$ and $tgt(p) = tgt(q)$* [25].

Definition 17. *A congruence on G is a relation $\simeq$ on $Path_G$ with the following* [25]*:*

1. *The relation $\simeq$ is an equivalence relation.*
2. *If $p \simeq q$, then $src(p) = src(q)$ and $tgt(p) = tgt(q)$.*
3. *If given paths $p, p\prime : a \rightarrow b$ and $q, q\prime : b \rightarrow c$, and if $p \simeq p\prime$ and $q \simeq q\prime$, then $(p + +q) \simeq (p\prime + +q\prime)$.*

Informally, a congruence is an enhanced equivalence relation that marks how different paths in G relate to one another by enforcing additional constraints; pairing a graph with a congruence forms a schema [25].

Categorical View of Schemas. We give Spivak's definition of a schema below; this definition is generic enough to also apply to faceted browsing when looking at the underlying graph of the facet categories. Figure 4 contains a schema corresponding to the medications example from Fig. 3.

Definition 18. *A schema S is a named pair $S = (G, \simeq)$, where G is a graph and $\simeq$ is a congruence on G* [25].

Note that the keys in Fig. 4 would normally be integer keys, but here text labels are applied to increase readability and to improve the ease of understanding the example. The resource table in this schema contains a generic list of resources (for example, documents or library items) where each resource has a foreign key indicating how it is classified. The medications table contains a list of classes and sub-classes for medications, as well as a self-referential foreign key pointing back at itself; this foreign key indicates this particular medication's ancestor. The self-referential key gives additional structure to the medication classes and sub-classes found within the table without the need for additional relationship tables; this method of storing a taxonomy is similar to closure tables [16].

In Fig. 4, the entry with *Medication* as its key has no foreign key. This null relationship indicates that it is the root of this particular facet graph; with respect to the category-theoretic model, it implies there are no morphisms having this object in its domain.

3.3 Instances of Facets and Faceted Taxonomies

An instance of a facet is a collection of objects whose data are classified according to specific relationships, such as the one illustrated in Fig. 3. We formalize this below using Spivak's instances of schemas as inspiration [25].

Definition 19. *Let $F = (U(\boldsymbol{Facet_i}), \simeq)$, where the graph underlying a facet type is denoted $U(\boldsymbol{Facet_i})$ for some $\boldsymbol{Facet_i} \in Ob(\boldsymbol{FacetTax})$ and where $\simeq$ is a congruence on $U(\boldsymbol{Facet_i})$. An instance on F is denoted $(Facet, Ancestor) : F \to$ **Set** where:*

1. *Facet is a function defined as $Facet : V \to$ **Set**, so for each vertex $v \in V$ we can recover a set of facets denoted $Facet(v)$ within this facet type.*
2. *for every arrow $a \in A$ having $v = src(a)$ and $w = tgt(a)$, a function $Ancestor(a) : Facet(v) \to Facet(w)$.*
3. *congruence is preserved: for any $v, v\prime \in V$ and paths p,$p\prime$ from v to $v\prime$ where $p = v[f_0, f_1, f_2, \ldots, f_m]$ and $p\prime = [f_0\prime, f_1\prime, f_2\prime, \ldots, f_n\prime]$, if $p \simeq p\prime$, for all $x \in Facet(v)$, $ancestor(f_m) \circ \ldots \circ ancestor(f_1) \circ ancestor(f_0)(x) = ancestor(f_n\prime) \circ \ldots \circ ancestor(f_1\prime) \circ ancestor(f_0\prime)(x) \in Facet(v\prime)$*

To create instances of **FacetTax**, the logic remains the same from **Facet**: take the underlying graph and a congruence. Instead of looking at the underlying graph of a single facet type, the underlying graph of **FacetTax** is considered. We will use instances in the next section to model the integration and reuse multiple heterogeneous sources of information.

4 Bootstrapping Faceted Taxonomies

Faceted taxonomies are common in the biomedical domain where controlled vocabularies are curated and integrated into interfaces in order to assist in the exploration and interaction required by the system. We present two different use cases for faceted taxonomies with different requirements: one where merging heterogeneous terminologies into a single taxonomy fits the design of the interface (for example, i2b2) and one where having control over multiple independent instances of facets is desired (for example, DELVE).

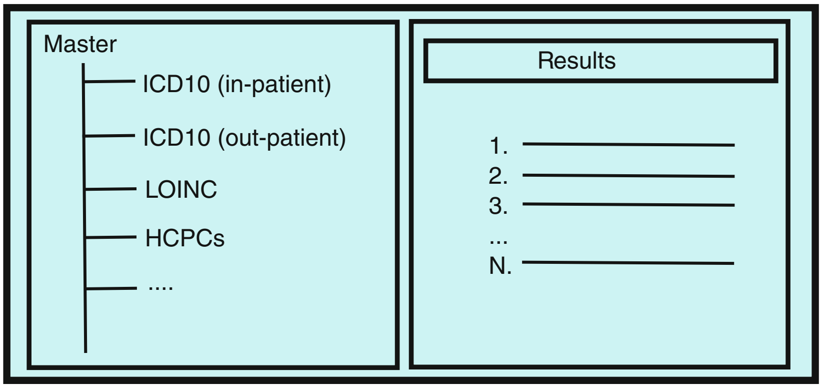


Fig. 5. A web interface could merge multiple instances together into a master taxonomy.

4.1 Designing Faceted Systems

A common design for faceted systems that require multiple terminologies is to simply merge everything together into a centralized master taxonomy; this merged taxonomy is often how lightweight ontologies, discussed as one of the three foundations of facet models [30], are constructed. The merged taxonomy may or may not have multiple instances of the same terminology, depending upon what is needed for the interface. For example, in the conceptual skeleton of the interface presented in Fig. 5, the merged taxonomy has multiple existing biomedical terminologies, including two instances of ICD10, based upon whether the resources are classified as belonging to in-patient or out-patient resources. In Sect. 4.2, we will discuss i2b2, a modern biomedical research tool that estimates patient cohort sizes by constructing Boolean queries from a merged faceted taxonomy.

Alternatively, multiple terminologies can peacefully co-exist within a single interface without being merged into a master taxonomy. In fact, it could be a pivotal design element in the interface that allows for a deeper exploratory search of the resources by enabling multiple points of faceted search. In Fig. 6, we show a conceptual skeleton for a faceted system utilizing multiple terminologies and multiple instances of ICD10. For example, such an interface could leverage ICD10 to draw a graph of facets (i_0) and a tree of related facets (i_2) and enable the user to interactively explore resources which could be a simple list with annotations (i_1). This example is similar to the spirit of DELVE, discussed in Sect. 4.8, where facets are contained within and help drive visualizations.

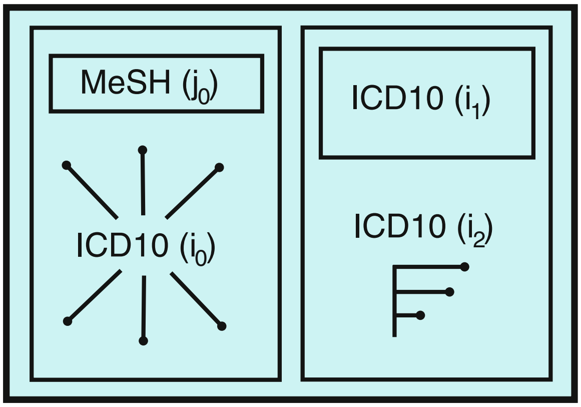


Fig. 6. A web interface containing multiple instances of a terminology in discrete components assists interaction.

4.2 i2b2

The i2b2 (Informatics for Integrating Biology and the Bedside) query tool allows researchers to locate patient cohorts for clinical research and clinical trial recruitment [19]; the tool itself provides a drag-and-drop method of creating Boolean queries of inclusion and exclusion criteria from a hierarchical list of facets. For example, if someone wanted to search for only female patients, they would click into the *Demographics* facet, into the *Gender* facet, and drag *Female* to the first query panel. In addition, if they wanted female diabetics, they would also navigate into the *Diagnoses* facet and drag the desired type of diabetes into the second panel. i2b2's Boolean queries are formed from having logical *or*-statements across panels and *and*-statements within a panel. With respect to the example above, if the user wanted female diabetic and hypertensive patients, they would also find the hypertension facet and drag it into the same panel having diabetes, so that the panel represents patients having either diabetes or hypertension. This Boolean construction can be continued with any number of facets from any number of terminologies.

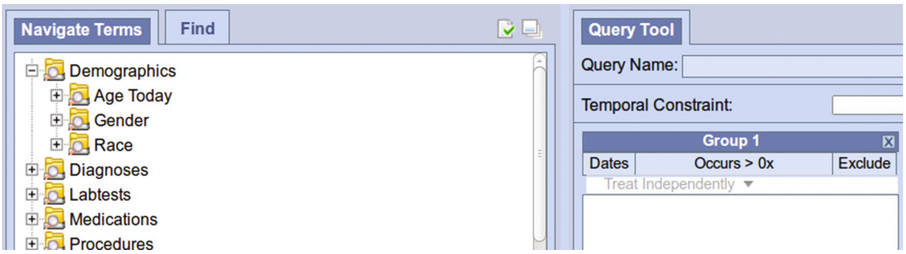


Fig. 7. The i2b2 query tool uses drag-and-drop interaction to construct patient queries.

The biomedical domain has a long history of curating and maintaining controlled vocabularies and terminologies, such as those found in the Unified Medical Language System (UMLS) [2]. The structure behind these terminologies is a rich source for building faceted browsing systems that explore resources having been classified with these standards.

In Fig. 7, the taxonomy of a local implementation of i2b2 is partially shown; note that every facet type of a patient is compiled into a central taxonomy as part of the meta-data cell for i2b2 [19]. This means that the central taxonomy has very different concepts, such as diagnoses and laboratory procedures, residing in the same table. Our local implementation of i2b2 uses ICD10 codes [31] for diagnoses and HCPCs codes [5] for procedures; these terminologies are externally and independently curated and made available by their creators. To i2b2, diagnosis is a facet type and ICD10 provides the organizational structure behind diagnoses, but ICD10 is a full terminology and one can consider ICD10 itself to be a facted taxonomy for diagnoses; the use of large-scale existing terminologies in faceted browsing system blurs the line between facet types and facet taxonomies, similar to our example and discussion of Fig. 3. Our modeling technique needs to be able to abstractly and consistently model both of these cases. In either case, the goal is encouraging the reuse of existing terminologies so that our faceted taxonomies contain accepted interoperable standards. An extension of i2b2 allows networking queries between institutions, so that one Boolean query can return counts of patients from multiple clinical sites; this would be impossible without integration of accepted biomedical terminologies into the faceted backbone of i2b2.

4.3 Merge Operations

Suppose we have multiple instances of facets, $I_0, I_1, \ldots, I_N$, how do we satisfy the requirements of an application such as i2b2 that expects a single instance to act as a master? For example, I_0 could be medications, while I_1 could be procedures, and so on.

Each $\mathbf{Facet}_i$ category is disjoint and contains no linkage to another $\mathbf{Facet}_j$ where $i \neq j$, so we must manufacture a link. This link is a meta-facet, an organizational tool that typically aids in drawing the faceted taxonomy [9]. By design, the meta-facet must connect to the root of each facet; we can easily identify the root in our facet graph because it is the only entry with a null ancestor. Given an instance, such as I_0 above, we know that the root of I_0 is the source of an arrow $a \in A$ from $U(\mathbf{Facet}_0)$ where $Ancestor(a)$ is the empty set; we shall call this function that returns the root object $root(I_i) : A \rightarrow \mathbf{Set}$ for some instance I_i.

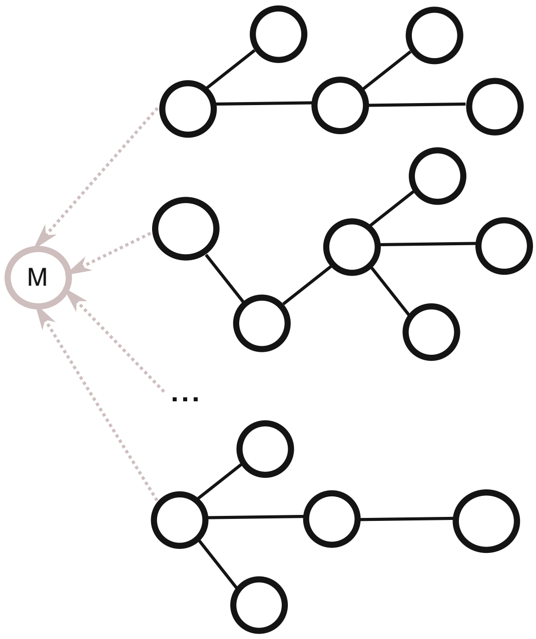


Fig. 8. A meta-facet can assist in merging facets together by providing a common anchor point.

Definition 20. *Let $\boldsymbol{Facet}_M$ be a meta-facet category for categories $\boldsymbol{Facet}_0, \ldots,$ $\boldsymbol{Facet}_N$, containing a meta-object and the roots of the others:*

$$Ob(\boldsymbol{Facet}_M) = M \cup root(I_0) \cup \ldots \cup root(I_N)$$

M is a meta-object sharing a relationship with every object: $Hom_{\boldsymbol{Facet}_M}(M, x)$ for each $x \in Ob(\boldsymbol{Facet}_M)$.

Figure 8 illustrates adding a meta-facet to join together a collection of facets; each black subtree represents a particular facet type. M is a new meta-object that must be created; the gray and dotted arrows that link this meta-object and the roots of the other facet graphs must be created as well.

Let us define the union of two underlying graphs, $U(\mathbf{Facet}_i)$ and $U(\mathbf{Facet}_j)$, as the union of its constituent parts. By definition, the sets of vertices and arrows for graphs underlying two **Facet** categories, $\mathbf{Facet}_i$ and $\mathbf{Facet}_j$, are disjoint and can be merged with the union of corresponding vertices and arrows; this leaves the graph disconnected, since $\mathbf{Facet}_i$ and $\mathbf{Facet}_j$ have no object in common.

Using the root of each instance and a meta-facet, we can create a new instance connecting every other underlying graph to our meta-facet:

Definition 21. *The merger of instances $I_0, I_1, \ldots, I_N$ of categories $\boldsymbol{Facet}_0, \ldots,$ $\boldsymbol{Facet}_N$ is a new instance I_M on $(G_U, \simeq_U)$ where:*

1. *$G_U = U(\boldsymbol{Facet}_0) \cup \cdots \cup U(\boldsymbol{Facet}_N) \cup U(\boldsymbol{Facet}_M)$. This is the union of the underlying graphs of the meta-data facet and the facets that are merging.*
2. *$\simeq_U$ is a congruence on G_U. We define this the same as in Sect. 3.3 but do note that the collection of paths have grown. No two paths in the merging categories conflict because the facets are disjoint by definition.*

The merged instance I_M is not defined much differently than $I_0, \ldots, I_N$ in that it still maintains $(Facet, Ancestor) : F \to \mathbf{Set}$ function mappings; the only difference is that the underlying graph has changed with additional path considerations. The merge operation is simply a transformation: we are manipulating the facets into a graph and symbolically merging graphs to suit our needs. The information regarding classified resources that is embedded into each facet gets reused; only the surrounding structure changes. In the following section, we formally define pullbacks and give an example of the utility of merged instances.

4.4 Pullback Operations

Recall that the objects of each $\mathbf{Facet}_i$ categories are sets of pointers toward resources which have been classified as belonging to a particular facet. Our model can create higher-level faceted groupings from existing facets by leveraging categorical pullback operations, also known as fiber products [25]; these operations model interactive conjunctions within instances of $\mathbf{Facet}_i$ and **FacetTax** categories, yielding new facet types that are not available directly in the taxonomy.

Definition 22. *Given sets A, B, $C \in Ob(\mathcal{C})$ for some category $\mathcal{C}$, a pullback of A and B over C is any set D where an isomorphism $A \times_C B \to D$ exists for $A \times_C B = \{(a, b, c) | f(a) = c = g(b)\}$; this is illustrated below, using Spivak's $\lrcorner$-notation to label the pullback* [25]*:*

$$\begin{tikzcd}
(a) & & B \arrow[d, "g"] & (b) & A \times_C B \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "g"] \\
& A \arrow[r, "f"'] & C & & A \arrow[r, "f"'] & C
\end{tikzcd}$$

The result of a pullback is easily illustrated with an example. If *horror* and *comedy* belong to the facet type for genres of either movies or books, then we can draw the relationships between *horror* and *comedy* easily:

$$\begin{tikzcd}
& Horror \arrow[d, "is"] \\
Comedy \arrow[r, "is"'] & Genre
\end{tikzcd}$$

We derive a new set that we can label *horror and comedy* by applying the pullback to the set of *horror* and the set of *comedy* objects:

$$\begin{tikzcd}
\text{Horror and Comedy} \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] \arrow[dr, phantom, "\lrcorner", very near start] & Horror \arrow[d, "is"] \\
Comedy \arrow[r, "is"'] & Genre
\end{tikzcd}$$

This forms a new set of objects being characterized by a conjunctive facet not directly found in the facet type; we could even give this new set a new semantic name: *comedic horror*. The direct semantic name for groupings found indirectly within the data can become an engaging element of the interface, eliminating the possibility of the user being limited only to interaction with facets defined directly within the original taxonomy.

The projection functions π_1 and π_2 may look trivial: a *comedic horror* title is clearly a comedy and clearly a horror title. Despite simplicity in appearance, the utility of the projection functions π_1 and π_2 mapping back to the original facets can be seen with faceted cues: for example, we can use π_1 to highlight *comedic horror* titles within the *horror* titles.

Since patients in i2b2 are classified across multiple merged terminologies, we can use pullbacks to create reusable conjunctions to bridge across facet types in instances of **FacetTax**. A common goal within i2b2 is to identity groups of patient cohorts by dragging and dropping facets from a master taxonomy. A clinical researcher can quickly refine Boolean queries targeting patient populations; often these queries have a base population that can be specified as a conjunction. For example, a clinical researcher studying patients with breast cancer who have undergone a mastectomy procedure needs the ability to quickly reference such a population. We diagram what the data provides below:

$$\begin{tikzcd} & Procedure(Mastectomy) \arrow[d, "\text{belongs to}"] \\ Diagnosis(BreastCancer) \arrow[r, "\text{belongs to}"'] & Patient \end{tikzcd}$$

If we apply the pullback to the category of *procedure (Mastectomy)* and the category of *diagnosis (Breast Cancer)*, we get a new category that we can label *Breast Cancer and Mastectomy*:

$$\begin{tikzcd} \text{Breast Cancer and Mastectomy} \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] \arrow[dr, phantom, "\lrcorner", very near start] & Procedure(Mastectomy) \arrow[d, "\text{belongs to}"] \\ Diagnosis(BreastCancer) \arrow[r, "\text{belongs to}"'] & Patient \end{tikzcd}$$

This new category becomes an interactive element that can be reused within the interface; within i2b2, conjunctions can be annotated with a friendly human-readable name and can be shared across users. In the next section, we will demonstrate that pushouts help construct new facets from disjunctions.

4.5 Pushout Operations

Our model can assist in computing ad-hoc facets that attempt to compensate for short-comings in either the terminologies involved or the underlying data. In this section, we define what a pushout operation computes given specific sets of resources.

Definition 23. *Given sets A, B, $C \in Ob(\mathcal{C})$ for some category $\mathcal{C}$, a pushout of sets B and C over A is any set D where an isomorphism $B \sqcup_A C \to D$ exists; this is illustrated below, using Spivak's $\ulcorner$-notation to label the pushout* [25]*:*

$$(a)\quad \begin{tikzcd} A \arrow[r, "g"] \arrow[d, "f"'] & C \\ B & \end{tikzcd} \qquad (b)\quad \begin{tikzcd} A \arrow[r, "g"] \arrow[d, "f"'] & C \arrow[d, "i_2"] \\ B \arrow[r, "i_1"'] & B \sqcup_A C \arrow[ul, phantom, "\ulcorner", very near start] \end{tikzcd}$$

It is important to note that $B \sqcup_A C$ was formed by the quotient of the disjoint union of A, B, C and an equivalence relation on B and C with A. An example will help demonstrate the utility of pushouts. Hypertensive patients are woefully under-diagnosed and relying solely on diagnosis codes to locate patients with hypertension is problematic [28]. In addition to diagnosis codes, vital signs are either recorded by medical providers or recorded by machines at given intervals; these measurements can be used to determine a person's hypertensive state [28]. Recall that our resources for i2b2 are patients; patients can have diagnoses (from an instance of the ICD10 terminology) and vitals signs (from an instance of the LOINC terminology):

$$\begin{tikzcd} Patient \arrow[r, "has"] \arrow[d, "has"'] & Diagnosis(Hypertension) \\ Vital(BP > 140/90) & \end{tikzcd}$$

We compute the pushout and receive a single anchor for those individuals that were either coded to have a hypertension diagnosis code or that were recorded having high blood pressure:

$$\begin{tikzcd} Patient \arrow[r, "has"] \arrow[d, "has"'] & Diagnosis(Hypertension) \arrow[d, "i_2"] \\ Vital(BP > 140/90) \arrow[r, "i_1"'] & \text{Hypertension or BP} > 140/90 \arrow[ul, phantom, "\ulcorner", very near start] \end{tikzcd}$$

The pushout acts as a convenient, derived facet that the user can interact with just like any other facet. In i2b2, the disjunction between diagnoses codes and vital signs can be performed without the pushout because the interface itself allows for Boolean queries to be performed by dragging and dropping any facet into its query window. The value of the pushout is simply for convenience and reuse: when the pushout is used by multiple people, the context of a patient with hypertension is made clear and reusable.

4.6 Faceted Views

We can construct commonly-used patient cohorts via pullback and pushout operations. In another example, we consider chronic kidney disease (CKD). CKD suffers from an issue similar to hypertension: diagnostic codes are not always used and might not capture the true disease state of the population, but laboratory results can predict CKD, including the disease's stage [17].

In Fig. 9, we show that pushouts can create new facets from existing ones in order to better address the needs of the interface. We create a facet for a

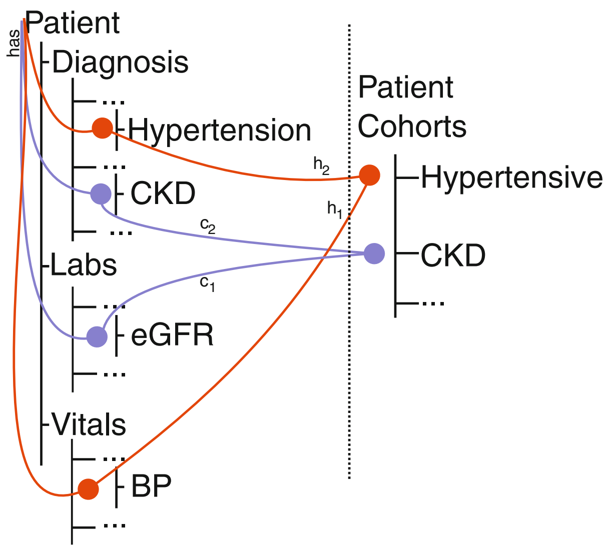
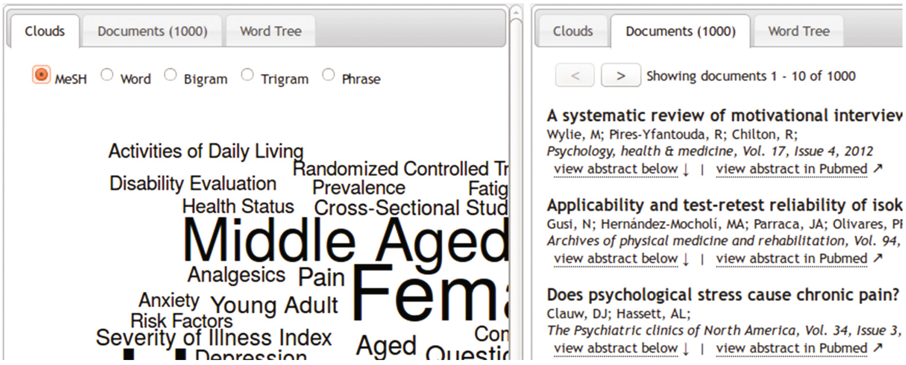


Fig. 9. Faceted views can provide convenient, derived facets for interactivity.

hypertensive cohort by the pushout of diagnostic codes for hypertension and qualifying vital signs; we also create a cohort of CKD patients by considering CKD diagnostic codes and qualifying eGFR lab results. d_1 and d_2 are inclusion maps for the hypertensive cohort, while c_1 and c_2 are inclusion maps for the CKD cohort. The underlying taxonomies for patient data are large, having up to tens of thousands of nodes. The ability to create faceted views on top of the standard taxonomy will greatly improve the usability of an interface by providing the user with the most meaningful and efficient facets, resulting in targeted and relevant resources.

4.7 Implementation

If we connect instances back to our notion that schemas are not structurally different than facets, it is clear that I_M is simply another table containing $N+1$ relationships with entries from the $\mathbf{Facet}_0, \ldots, \mathbf{Facet}_N$ categories sharing a relationship with the meta-facet. The foreign keys of these meta-relationships would simply point back to the roots of the other facets; this enables reuse in-place without needlessly copying data. Furthermore, this gives a clear implementation path for enabling reusable terminologies in a standard relational database, where tables help structure facets and the resources that have been classified accordingly. If a relational database is not possible for the application, then an equivalent scheme can be mimicked in other environments. For example, a web-application could use JSON (Javascript Object Notation) data interchange format [3] to store the taxonomy and links to resources.



Showing documents 1 - 10 of 1000

A systematic review of motivational interviev
Wylie, M; Pires-Yfantouda, R; Chilton, R;
Psychology, health & medicine, Vol. 17, Issue 4, 2012
view abstract below ↓ | view abstract in Pubmed ↗

Applicability and test-retest reliability of isok
Gusi, N; Hernández-Mocholí, MA; Parraca, JA; Olivares, Pl
Archives of physical medicine and rehabilitation, Vol. 94,
view abstract below ↓ | view abstract in Pubmed ↗

Does psychological stress cause chronic pain?
Clauw, DJ; Hassett, AL;
The Psychiatric clinics of North America, Vol. 34, Issue 3,
view abstract below ↓ | view abstract in Pubmed ↗

Fig. 10. DELVE contains visualizations controlled by facets as well as visualizations that contain facets.

4.8 DELVE

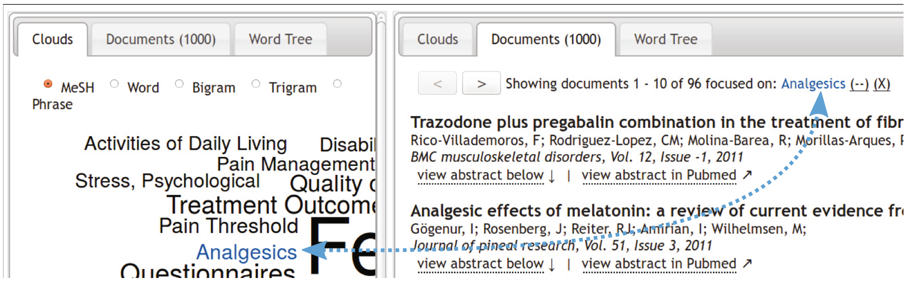
DELVE (Document ExpLoration and Visualization Engine) is our framework and application for browsing biomedical literature through heavy use of visualizations [12,13]. In fact, our motivation for choosing category theory began when first designing DELVE, due to the difficulty in modeling facets that are controlled by visualizations or found within a visualization. In the case of i2b2, the design of the interface insists on merging terminologies together into a master taxonomy that directs exploration within the interface. With DELVE supporting multiple visualizations, a master taxonomy is unrealistic as each visualization potentially requires a different set of facets altogether.

Understanding DELVE. In Fig. 10, a query for fibromyalgia is shown. The screen is split into two parts for this example; the abbreviated left-hand side contains a cloud [27] and the right-hand side contains a list of relevant biomedical publications. The default cloud shows the frequency of terms using the MeSH (Medical Subject Headings) vocabulary; librarians at the National Library of Medicine manually review journal articles and tag them with appropriate MeSH terms [18]. MeSH terms are hierarchically organized and are typically accurate reflections of the article's contents since they are manually assigned, making them great facet candidates. In addition to MeSH terms, we extend the general concept of world clouds [27] to unigrams, bigrams, trigrams, and common phrases.

DELVE provides other collections of terms as facets for two reasons: (1) interdisciplinary collaboration typically involves researchers interested in biomedical literature who are not familiar with MeSH terms and (2) granularity and phrasing of terms can be an issue. For example, a researcher queries for fibromyalgia using DELVE as seen in Fig. 10; they are also interested in *functional somatic syndromes* but this term is not directly available as a MeSH term. Instead, articles covering *functional somatic syndromes* are typically tagged *somatoform disorders*; without this knowledge, a researcher could miss desired articles. DELVE resolves this issue by providing a list of biomedical trigrams as a facet, which was compiled by analyzing all trigrams found within Pubmed's library of biomedical articles; the phrase *functional somatic syndromes* occurs in great frequency. From a modeling perspective, there are natural differences in the structure of the MeSH hierarchy and the collection of anchoring trigrams, but our categorical model naturally accounts for this by allowing objects to have any inclusive relationship within **Facet** categories: including those who have many (MeSH terms) and those who have none (DELVE's trigrams). In DELVE's case, instances of facets play a role when creating focused collections of documents based on what the user has selected through the interface, which could potentially span one or more facets.

Other visualizations, such as word trees and histograms, are available as part of the extensible nature of DELVE. We give an example of MeSH clouds and word trees working together in Sect. 4.9.

Focusing Considerations. The annotated screen-shot in Fig. 11 demonstrates DELVE's ability to use a facet to focus. In this example, a search for fibromyalgia is focused on the MeSH term *analgesics*, which causes the documents viewer to show only those documents that are classified as belonging to the MeSH term



Showing documents 1 - 10 of 96 focused on: Analgesics (--) (X)

Trazodone plus pregabalin combination in the treatment of fibr
Rico-Villademoros, F; Rodriguez-Lopez, CM; Molina-Barea, R; Morillas-Arques, I
BMC musculoskeletal disorders, Vol. 12, Issue -1, 2011
view abstract below ↓ | view abstract in Pubmed ↗

Analgesic effects of melatonin: a review of current evidence fr
Gögenur, I; Rosenberg, J; Reiter, RJ; Amirian, I; Wilhelmsen, M;
Journal of pineal research, Vol. 51, Issue 3, 2011
view abstract below ↓ | view abstract in Pubmed ↗

Fig. 11. A DELVE search for fibromyalgia publications focusing on analgesics

analgesics. Multiple points of focus are supported in the subsequent version of DELVE, such as focusing using different word clouds [27] and word trees [29]. If the user also selects the MeSH term *female*, the document viewer would only show those documents tagged with both MeSH terms *analgesics* and *female*. Color is used to visually offset the facets being focused upon. The document viewer ranks results according to how many occurrences of the focus terms can be found within the abstract of the corresponding article.

Within one faceted taxonomy, aggregating focuses becomes a focused version of the queries discussed in Sect. 2. Suppose the user also wishes to focus on the trigram *functional somatic disorders*. If we have created instances of **Facet** categories as discussed in Sect. 3.3, we can also create instances of focused subcategories by taking a subgraph of the graph underlying **Facet**:

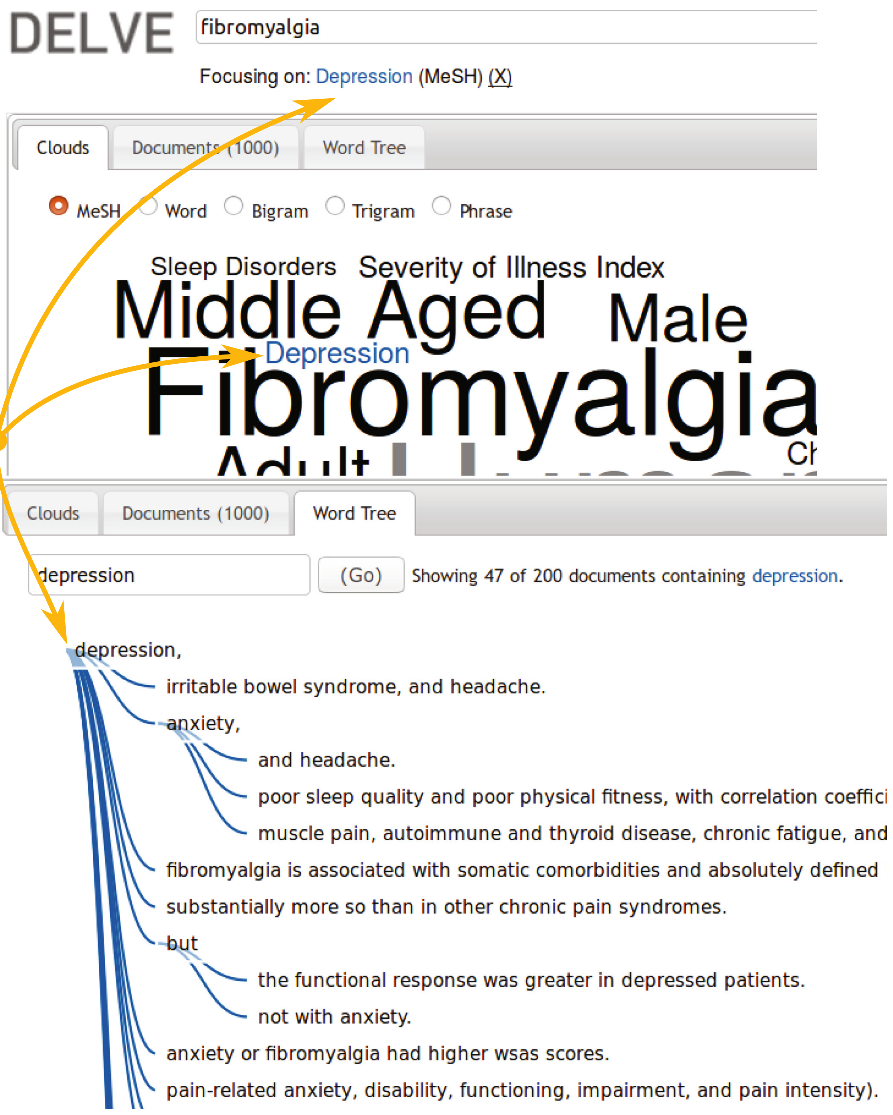
Definition 24. *Given instances $I_0, I_1, \ldots, I_N$ of categories $\textbf{\textit{Facet}}_0, \ldots, \textbf{\textit{Facet}}_N$, let $I_{F0}, I_{F1}, \ldots, I_{FN}$ be focused instances created by replacing $U((\textbf{\textit{Facet}}_i))$ with $U(\textbf{\textit{Focus}}_i)$ for $i = 0, 1, \ldots, N$.*

Recalling Resources. At some point during a user's interactive session in a faceted browsing system, it is advantageous or desirable to recall and list all resources that were classified according to a focused selection of facets. When creating instances of our facet categories, we defined a function capable of returning the ancestor of the facet type for a given facet. We can similarly define a function capable of returning focused resources.

Definition 25. *Let R be a function defined as $R(Focus, Resource) : \textbf{\textit{Focus}} \rightarrow \textbf{\textit{Set}}$, where:*

1. *Focus is a function similar to the Facet defined in Sect. 3.3: $Focus : V \rightarrow \textbf{\textit{Set}}$, so for each vertex $v \in V$ we can recover a set of focused facets denoted $Focus(v)$*
2. *Resource is a function defined for every focused facet $f \in Focus(v)$ above as $Resource(f) : Focus(v) \rightarrow Resource(f)$.*

In other words, similar to how we defined a function *Ancestor* in Sect. 3.3 as a self-referential link back to facets, we now define a function that unrolls the foreign relationship between facets and resources. An example of this is seen in Fig. 4: the resource with *resource 2* as its key holds a foreign relationship with the medication that has *anti-diabetic* as its primary key. Relating this back to the definition above, we rephrase this as: for every facet in the graph, collect their primary keys (PKs) and from the resource table, collect any primary keys where any foreign keys match the original keys (PKs). At this point, the interface is free to present the resources



Showing 47 of 200 documents containing depression.

depression,
irritable bowel syndrome, and headache.
anxiety,
and headache.
poor sleep quality and poor physical fitness, with correlation coeffici
muscle pain, autoimmune and thyroid disease, chronic fatigue, and
fibromyalgia is associated with somatic comorbidities and absolutely defined
substantially more so than in other chronic pain syndromes.
but
the functional response was greater in depressed patients.
not with anxiety.
anxiety or fibromyalgia had higher wsas scores.
pain-related anxiety, disability, functioning, impairment, and pain intensity).

Fig. 12. A DELVE search for fibromyalgia publications focusing on depression

as needed, which consequentially allows us to model ranking and sorting schemes for resources; we leave these discussions as future work.

4.9 Interacting with Word Clouds and Trees

In DELVE, facets contained in visualizations can work together harmoniously through a centralized point of focus; by default, focusing in one visualization will set the focus in all other visualizations. In Fig. 12, we show a DELVE search for fibromyalgia and the result of focusing on the MeSH term *depression*. Within the MeSH cloud, the term is highlighted with blue and a secondary reminder cue containing the focused term is placed below the original query. The word tree redraws itself with the selected term as the root of the tree; this shows occurrences of the term *depression* within the sentences belonging to the classified resources, where redundancy is collapsed to a common prefix. For example, the following phrases are collapsed under tree nodes for *depression* followed by *anxiety*:

1. *depression, anxiety, and headache.*
2. *depression, anxiety, poor sleep quality and poor physical fitness...*
3. *depression, anxiety, muscle pain, autoimmune and thyroid disease...*

From Fig. 12, we can also see that the phrases of the form {*depression, anxiety, and ...*} and the phrase {*depression, but not with anxiety*} point to different resources containing relationships between *fibromyalgia*, *depression*, and *anxiety*. The goal of DELVE is to immerse a researcher into an exploratory search system where visualizations help expedite the discovery process. This goal is made easier by constructing DELVE upon a solid theoretical foundation that has been demonstrated to intelligently reuse and integrate existing biomedical terminologies.

5 Future Work

As mentioned previously, a natural consequence of modeling facets, faceted taxonomies, and faceted browsing systems is that resources ultimately are retrieved. This opens the door to abstractly modeling and developing deeper manipulations of faceted data in a way that is transparent and reusable across systems. For example, we demonstrated that categorical constructions such as pullbacks and pushouts can help dynamically organize and reorganize faceted data. These types of operations could potentially lead to creating facets dynamically, where new facets are created on the fly from computations involving existing facets. Other operations, such as retractions, need to be explored so that their role in the model is fully understood; this is the groundwork toward the next steps of ranking and sorting resources. It is important to note that category theory focuses largely on structure, but structural similarity does not necessarily imply functional similarity; existing knowledge bases and terminologies must be intelligently used and reused to supplement and extend our abstract framework.

We are developing an application programming interface (API) for faceted browsing and wish to include support for interfaces that require multiple heterogeneous terminologies. The mapping between schemas and facets clears the path to implementation with a database containing faceted data and taxonomies. Support for functional databases is growing [23,24], but a traditional relational database is adequate. An API for faceted browsing can bridge the gap between a categorical model for faceted browsing and databases, allowing us to start with traditional relational databases and migrate towards functional databases as they mature.

The impact that visualizations play in faceted browsing systems deserves to be explored further. In systems such as DELVE, one interaction can have consequences in many parts of the interface. Ultimately, with a categorical model, one will be able to mathematically prove something is possible before implementation; the relationships and road maps between proof and implementation paths need to be researched further.

6 Conclusions

We extended our category-theoretic model of faceted browsing to support multiple heterogeneous terminologies as facets, which are needed in interfaces where more than one source of information controls the exploration of the data. Two use-cases emerged from our discussions of integrating multiple terminologies: merging instances into a single master and operation considerations when managing multiple facets.

We also showed that facets are categorically similar to database schemas, which allowed us to create instances of facets and faceted taxonomies and in turn support modeling heterogeneous terminologies as facets. Our model was previously demonstrated to encourage the reuse and interoperability of existing facet models [9], but the additional extensions presented also encourage the reuse of existing terminologies and provide a clear path to integrating them as controllable facets within a faceted browsing system.

Acknowledgment. The project described was supported by the NIH National Center for Advancing Translational Sciences through grant numbers UL1TR001998 and UL1TR000117. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.

References

1. Barr, M., Wells, C.: Category Theory for Computing Science. Prentice Hall, New York (1990)
2. Bodenreider, O.: The unified medical language system (UMLS): integrating biomedical terminology. Nucleic Acids Res. **32**(suppl. 1), D267–D270 (2004)
3. Bray, T.: The Javascript Object Notation (JSON) data interchange format (March 2014), https://tools.ietf.org/html/rfc7159/, retrieved 15 June 2016

4. Chu, H.J., Chow, R.C.: An information model for managing domain knowledge via faceted taxonomies. In: 2010 IEEE International Conference on Information Reuse and Integration (IRI), pp. 378–379. IEEE (2010)
5. CMS: Healthcare Common Procedure Coding System (HCPCS). Centers for Medicare & Medicaid Services (2003)
6. Coecke, B., Paquette, É.O.: Categories for the practising physicist. In: New Structures for Physics, pp. 173–286. Springer (2011)
7. Dawson, A., Brown, D., Broughton, V.: The need for a faceted classification as the basis of all methods of information retrieval. In: Aslib Proceedings: New Information Perspectives, vol. 58, pp. 49–72. Emerald Group Publishing Limited (2006)
8. Fagan, J.C.: Usability studies of faceted browsing: a literature review. Inf. Technol. Libr. **29**(2), 58–66 (2013)
9. Harris, D.R.: Modeling reusable and interoperable faceted browsing systems with category theory. In: 2015 IEEE International Conference on Information Reuse and Integration (IRI), pp. 388–395. IEEE (2015)
10. Harris, D.R.: Foundations of reusable and interoperable facet models using category theory. Inf. Syst. Front. **18**(5), 953–965 (2016)
11. Harris, D.R.: Modeling integration and reuse of heterogeneous terminologies in faceted browsing systems. In: 2016 IEEE 17th International Conference on Information Reuse and Integration (IRI), pp. 58–66. IEEE (2016)
12. Harris, D.R., Kavuluru, R., Jaromczyk, J.W., Johnson, T.R.: Rapid and reusable text visualization and exploration development with delve. In: Proceedings of the Summit on Clinical Research Informatics. AMIA (2017)
13. Harris, D.R., Kavuluru, R., Yu, S., Theakston, R., Jaromczyk, J.W., Johnson, T.R.: Delve: A document exploration and visualization engine. In: Proceedings of the Summit on Clinical Research Informatics, p. 179. AMIA (2014)
14. Hearst, M.A.: Clustering versus faceted categories for information exploration. Commun. ACM **49**(4), 59–61 (2006)
15. Hearst, M.A.: Design recommendations for hierarchical faceted search interfaces. In: SIGIR Workshop on Faceted Search, pp. 1–5. ACM (2006)
16. Karwin, B.: SQL Antipatterns: Avoiding the Pitfalls of Database Programming, 1 edn. Pragmatic Bookshelf (2010)
17. Kern, E.F., Maney, M., Miller, D.R., Tseng, C.L., Tiwari, A., Rajan, M., Aron, D., Pogach, L.: Failure of icd-9-cm codes to identify patients with comorbid chronic kidney disease in diabetes. Health Serv. Res. **41**(2), 564–580 (2006)
18. Lowe, H.J., Barnett, G.O.: Understanding and using the medical subject headings (mesh) vocabulary to perform literature searches. J. Am. Med. Assoc. **271**(14), 1103–1108 (1994)
19. Murphy, S.N., Weber, G., Mendis, M., Gainer, V., Chueh, H.C., Churchill, S., Kohane, I.: Serving the enterprise and beyond with informatics for integrating biology and the bedside (i2b2). J. Am. Med. Inform. Assoc. **17**(2), 124–130 (2010)
20. Niu, X., Hemminger, B.: Analyzing the interaction patterns in a faceted search interface. J. Assoc. Inf. Sci. Technol. (2014), http://dx.doi.org/10.1002/asi.23227
21. Phillips, S., Wilson, W.H.: Categorial compositionality: a category theory explanation for the systematicity of human cognition. PLoS Comput. Biol. **6**(7), e1000858 (2010)
22. Prieto-Díaz, R.: A faceted approach to building ontologies. In: IEEE International Conference on Information Reuse and Integration, IRI 2003, pp. 458–465. IEEE (2003)
23. Spivak, D.I.: Simplicial databases. arXiv preprint arXiv:0904.2012 (2009)

24. Spivak, D.I.: Functorial data migration. Inf. Comput. **217**, 31–51 (2012)
25. Spivak, D.I.: Category Theory for the Sciences. MIT Press, Cambridge (2014)
26. Spivak, D.I., Giesa, T., Wood, E., Buehler, M.J.: Category theoretic analysis of hierarchical protein materials and social networks. PLoS ONE **6**(9), e23911 (2011)
27. Viegas, F.B., Wattenberg, M., Feinberg, J.: Participatory visualization with wordle. IEEE Trans. Visual Comput. Graphics **15**(6), 1137–1144 (2009)
28. Wall, H.K., Hannan, J.A., Wright, J.S.: Patients with undiagnosed hypertension: hiding in plain sight. JAMA **312**(19), 1973–1974 (2014)
29. Wattenberg, M., Viégas, F.B.: The word tree, an interactive visual concordance. IEEE Trans. Visual Comput. Graphics **14**(6), 1221–1228 (2008)
30. Wei, B., Liu, J., Zheng, Q., Zhang, W., Fu, X., Feng, B.: A survey of faceted search. J. Web Eng. **12**(1–2), 41–64 (2013)
31. WHO: The ICD-10 classification of mental and behavioural disorders: clinical descriptions and diagnostic guidelines. In: The ICD-10 Classification of Mental and Behavioural Disorders: Clinical Descriptions and Diagnostic Guidelines. World Health Organization (1992)

A Layered Approach to Specification Authoring, Sharing, and Usage

John L. Singleton and Gary T. Leavens(✉)

Department of Computer Science, University of Central Florida, Orlando, USA
{jls,leavens}@cs.ucf.edu
http://methods.cs.ucf.edu

Abstract. Compositional reuse of software libraries is important for productivity. To promote reliability and correctness, there must also be a way to compose specifications for reuse. However, specifications cannot be adapted by the use of wrappers in the same ways as code, which leads to specifications being copied and modified. This copying and modification of specifications leads to poor maintainability and technical debt. We propose a system, Spekl, that solves these problems and makes compositional reuse of specifications possible in a way independent of the choice of specification languages and tools. We provide a detailed description of our system as well as provide details on our domain specific language for creating new tools, provide details on how to author new specifications, and demonstrate how Spekl facilitates compositional reuse through specification layering.

Keywords: Specification · Specification languages · JML · Reuse · Specification management · Spekl

1 Introduction

Software libraries have become a critical component of modern software engineering practice since they promote modularity and reuse. Unfortunately, the same cannot be said for specifications. This is because software libraries are typically reused in one of two ways: directly or adaptively. In *direct* reuse, the library matches the desired functionality perfectly and is simply reused as is. In *adaptive* reuse, the developer writes code that modifies (wraps) the library's computation in a way that makes it usable within the program at hand.

However, software specifications are encumbered by several issues that impact their reuse in ways that do not impact library code (we detail these differences in Sect. 2). Because of these differences, existing tools are inadequate for specification authoring and distribution.

The root of the problem is that, unlike software libraries, specifications may not always be compositionally adapted by their clients (we explain several different cases in which adaptation is made difficult in Sect. 2). This difference often

© Springer International Publishing AG 2018
S.H. Rubin and T. Bouabana-Tebibel (eds.), *Quality Software Through Reuse and Integration*, Advances in Intelligent Systems and Computing 561,
DOI 10.1007/978-3-319-56157-8_8

means that the specification must be copied and modified, a non-compositional form of reuse that is considered unacceptable for code. Directly modifying a copy (rather than using an adapter) negatively impacts modularity and makes it difficult to update to newer versions when they become available; hence such copy and modify adaptation is not used for code libraries. However, in the case of specifications, since an adapter is not always possible to create, the client may be forced to use the copy and modify technique, despite the maintenance problems it creates. The problem is that after modifying the copied specifications, the task of merging in ongoing work on the specification and keeping the local modifications becomes part of the technical debt of the client.

When using specifications, clients often have an existing code base that they want to check for some properties, such as safety or security. Verification of these properties must often use specifications for various libraries which are used in the code. If these library specifications are not appropriate for the verification, then they need to be adapted or modified. Furthermore, the code that clients want to check should not be modified to accommodate these specifications. Therefore, techniques that are based on code adaptation will not work.

We propose a system, Spekl,[1] aimed at solving this problem by making the software specification workflow easier to manage. Spekl addresses two key research questions:

RQ 1. How can tooling promote the reuse of pre-existing specifications?
RQ 2. How can tooling make existing specifications more useful?

The design of Spekl is based around the hypothesis that the problem of effectively solving the specification reuse problem can be reduced to three subproblems: tool installation/usage, specification consumption, and specification authoring and distribution. Before proceeding, we emphasize that the focus of Spekl is on externalized specifications, i.e., those separate from the text of the program. Spekl does not currently provide support for internalized specifications. This is because externalizing dependencies promotes modularity and thus facilitates reuse, both of which are goals of the Spekl system.

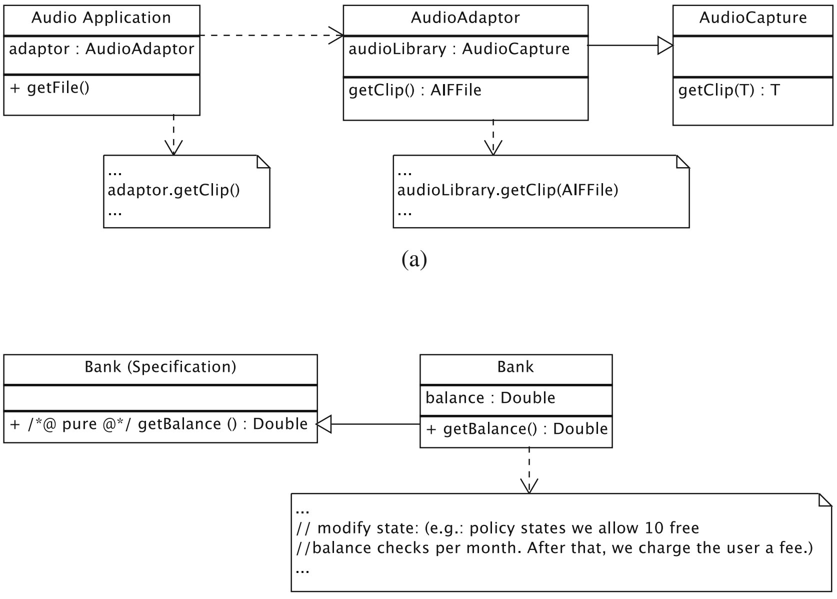
The remainder of this paper is organized as follows: In Sect. 2 we consider more deeply the motivation behind creating a tool tailored to the task of specification management. Then, in the following subsections, we explore these subproblems and explain how they relate to the design of the Spekl system. In Sect. 3 we review the main features of the Spekl system including verification tool installation and usage, specification consumption, and specification layering and authoring. In Sect. 4 we cover the details of creating new tools, provide a description of our domain specific language for writing new tools, and finally we provide details about publishing and installing tools. In Sect. 5 we discuss Spekl's features for consuming specifications and provide details on how the specification consumption features integrate with Spekl's tool running features. In Sect. 6 we discuss the specification authoring features of Spekl, including the

[1] The name *Spekl* comes from the abbreviation of Spekkoek, a cake originating in the Netherlands that is densely layered.

details on creating new specifications as well as layering specifications. In Sect. 7 we provide a discussion of the work related to the Spekl system and finally in Sect. 8 we conclude.

2 Problems in Specification Reuse

To further motivate our approach, in this section we examine specific problems that impact the reuse of specifications.


Audio Application
adaptor : AudioAdaptor
+ getFile()
...
adaptor.getClip()
...
AudioAdaptor
audioLibrary : AudioCapture
getClip() : AIFFile
...
audioLibrary.getClip(AIFFile)
...
AudioCapture
getClip(T) : T
(a)
Bank (Specification)
+ /*@ pure @*/ getBalance () : Double
Bank
balance : Double
+ getBalance() : Double
...
// modify state: (e.g.: policy states we allow 10 free
//balance checks per month. After that, we charge the user a fee.)
...

(b)

Fig. 1. (a) An example of an adapter that adapts the type of audio format produced by an audio capture library. (b) An example of attempting to adapt a pure method specification in an implementation that violates the purity.

Since specifications are associated with code, one might think that they could be managed using a source control system in the same way as code. However, in this section we will explain the unique aspects of working with specifications that make the direct use of such systems an incomplete solution to research questions 1 and 2.

As discussed in the introduction, the most fundamental difference between specifications and code is found in the kinds of changes one must make to each kind of artifact in order to perform adaptive reuse. For adaptive reuse of code, one can reuse existing source code without changing it by writing an adapter, e.g., a method that calls methods in an existing library to perform some task that the library does not support completely. This adaptation can be done in a purely additive manner, e.g., by adding new modules (containing new methods) that reuse existing code. For an example of this type of adaptation, see Fig. 1a. In the UML diagram, we depict an audio application that makes use of an audio library and an adapter to produce values in a format compatible with the client program.

However, for adaptive reuse of specifications, one may need to make semantic changes to the existing specifications. Although some changes are purely adaptive, such as adding missing specifications, others will not be.

2.1 Purity

Some examples where purely adaptive reuse does not work are caused by assumptions that verification tools must make for soundness; for example, a specification language (like JML [12]) that checks methods for lack of side effects may require that such methods must be declared to be "*pure*"; methods without such a declaration are assumed to not be pure, which is the only sound assumption, because only methods declared as pure are checked for lack of side effects. (In particular a pure method may only call other pure methods.) Thus, there is no way to write an adapter if one needs to prove that a particular method is pure; a new method that is declared to be pure cannot simply call the existing (non-pure) method, as that would not verify. The only fix is to change the specification of the existing method and verify its implementation; that is, to change the existing specification in a non-additive way. For an example of this type of problem, see Fig. 1b. In this example we show a specification for a banking application wherein a method has been marked "*pure*" in its specification. However, due to some new business requirements, a programmer must add a new behavior to the banking application that places a restriction on the number of free balance checks per month. To achieve this, purity must necessarily be violated since this method will have side effects, i.e., not all invocations of the method will produce the same results.

2.2 Specifications Are Not "One Size Fits All"

Why can't we simply create specifications that do not suffer from the problems described previously?

The task of creating a single specification to handle the needs of every conceivable user is daunting. This problem is rooted in the very nature of specification; that is, the goal is to specify a particular behavior that can be relied on. Any deviations from the base specification can cause it to become invalid and therefore unusable for verification.

To gain an intuitive understanding of the problem we will now consider an example which will not require knowledge of program verification techniques.

Consider the program and specification in Listing 1.1. This example was written in the JML [12] specification language and it specifies the behavior of a very simple function that adds two positive integers.

Listing 1.1. A simple program and specification.

```
public class MaybeAdd {
   //@ requires 0 < a;
   //@ requires 0 < b;
   //@ ensures  0 < \result;
   public static int add(int a, int b){
       return a+b;
   }
   public static void main(String args[]){
       System.out.println(add(2,3));
   }
 }
```

So what is the problem in Listing 1.1? The problem is as follows: the specification in Listing 1.1 only considers integers greater than zero. This is fine, but what if the integers are very large? An overflow condition is possible, though whether it is encountered by a runtime assertion checker depends on the test data used. A static checker however will detect this flaw and insist that the conditions in Listing 1.1 are made stronger. The code in Listing 1.2 shows the changes that are needed. Due to the changes made, the specification in Listing 1.2 does not refine the one given in Listing 1.1.

Listing 1.2. The updated specification of Listing 1.1. Note that the preconditions have been strengthened and the bounds on a and b have been tightened.

```
public class MaybeAdd {
   //@ requires 0 < a && a < Integer.MAX_VALUE/2;
   //@ requires 0 < b && b < Integer.MAX_VALUE/2;
   //@ ensures  0 < \result;
   public static int add(int a, int b){
       return a+b;
   }
   public static void main(String args[]){
       System.out.println(add(2,3));
   }
 }
```

2.3 Changes Invisible at the Language Level Can Cause Incompatible Specifications

In the previous sections we explained how specifications may be difficult to manage, because unlike source code, adaptation is not always possible, e.g., the method purity example in Sect. 2.1 and how writing a general specification to cover all possible use cases is difficult, e.g., the task of writing a specification flexible enough to be used effectively in runtime and static checking in the previous section. In this section, we discuss another problem that impacts specification reuse that has impacted a real-world project, OpenJML.

When using OpenJML, a user is expected to supply the specifications they want to check. To reduce programmer effort, OpenJML distributes a set of specifications for Java itself; without these specifications, any client that makes use of the Java standard libraries would be required to provide these specifications before checking their program.

Typically, between releases of Java, new library methods are added or modified. To accommodate this, the OpenJML project maintains specifications for

Java 4 thru Java 7. In order to modularize the effort, these specifications are maintained separately then combined to target a particular Java release. For example, the specifications for Java 7 would be omitted for an OpenJML release targeted for Java 6.

However, in preparation for the release of Java 8, the signature of the constructor for the `Class` class was changed between Java 1.7.79 and Java 1.7.80 to include an argument to the private constructor (this was missing in all prior versions of Java).

In the case of a library, this small change would have presented no problem, i.e., private constructors are hidden, furthermore, most client code will never directly construct a `Class` object. However, in the case of a specification, the prior specification (which did not include a parameter in the constructor) is therefore invalid if a JDK higher than 1.7.79 is used. To solve the problem, a user must either manually override the specification to be compatible with their JDK or rely on a special release of OpenJML that includes the specification of the `Class` class for that particular version of Java. The main difference between the two solutions is mainly that in the first case, the burden is on the user, and in second, an additional burden is placed on the tool author.

2.4 Subverting Security Specifications

Not all specifications are strictly behavioral in nature. Some specifications, such as those designed to describe information flow [2,7], are written specifically to prevent adaptation; if such adaptation were possible, it would not be possible to soundly enforce the security properties described by the library specifications. For example, consider an application designed to run on the Android operating system. A specification designed for use in an information flow verification tool may state that all values coming from the network should be treated as "untrusted." If an application utilizing the Android API has different security requirements and perhaps instead trusts network input, without copying and modifying the underlying specification, any calls to methods that produce values originating from the network will necessarily produce "untrusted" values. Rather than using the copy and modify approach, a more ideal solution would be to rely on tooling to remember the differences between the base specification and the enhancements made for the particular application.

3 Overview of Spekl's Features

Although the question of how to promote specification reuse is central to the design and motivation of Spekl, several other features such as support for installing and using tools are needed to support this goal. In the next three sections we will describe the three central features of the Spekl system: tool installation/usage, specification consumption, and finally, we will provide a description of the specification authoring and distribution features of Spekl. We provide an outline of these features in this section and provide additional details in Sects. 4–6.

3.1 Verification Tool Installation and Usage

In order to verify a program, a user must often install appropriate verification tools. However, due to the variety of tools and technologies used to author these tools, there does not exist a general way to go about installing them. This presents a problem not only in software engineering contexts [6,13], but also in teaching scenarios, where tools must be installed onto the computers of students. To that end, the first problem Spekl solves is the problem of distributing program verification software. Furthermore, since many verification tools such as OpenJML [3], SAW [4], and SPARTA [7] require external software packages, such as SMT solvers, in order to function properly, a single download is not sufficient to create a fully functioning verification environment.

Dependency management is a well-studied problem; on systems such as Linux, a user may choose to consume software via a variety of package management systems such as APT or YUM. While there are some cross-platform package management systems in existence, none have gained wide-spread usage and are therefore more likely to be detrimental to the purpose of increasing adoption of specification technologies.

Installing executable tools is only part of the problem. After a package is installed (even if a cross platform package manager were available) the user would then have to learn a series of tool-specific commands (or GUI actions) necessary to verify a piece of software. However, such specialization is not necessary. Spekl differs from a standard package management system in that Spekl provides an abstraction layer and a uniform interface to program verification tasks. Once a user configures project for Spekl, a user may run their verification tools by using the `spm` command from within the project directory.

The general command a user may use for checking their programs is the `spm check` command as shown in the following example:

```
~/my-project$ spm check
```

Executing the `spm check` command in this way causes all of the configured checks to be executed and their output displayed to the user. The uniform interface to running verification tools makes Spekl especially well-adapted for use in automated environments such as Continuous Integration.

All Spekl tool and specification packages are configured via a `package.yml` file, which is discussed in more detail in Sect. 4.2. Since the tool management aspect of Spekl is essentially that of a package manager, we provide all the usual support for tasks like dependency management, platform specific installation tasks, and packaging. Since these features are somewhat common to many systems, we omit their discussion in this paper for brevity.

3.2 Specification Consumption

Any non-trivial program is comprised of more than the source code written by one individual author. For example, if a user is writing an application with a graphical user interface, their application may rely on dozens of interdependent

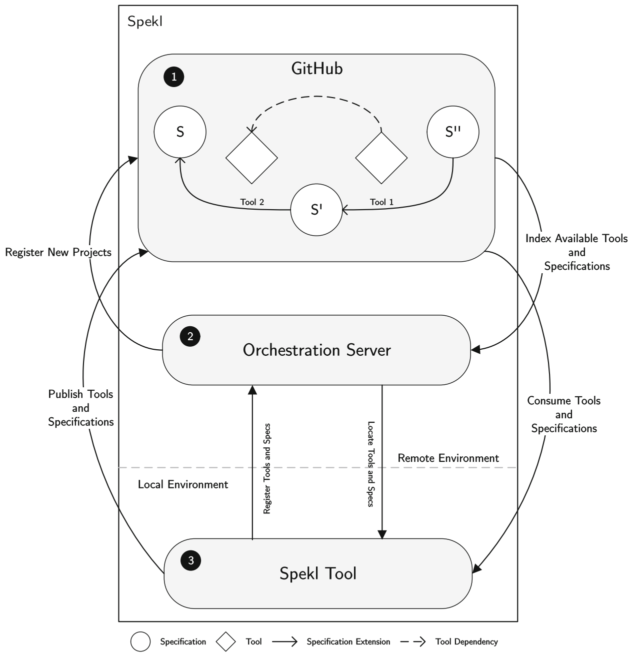


Fig. 2. An overview of the Spekl system. Spekl relies on a vertical architecture in which a decentralized repository of specifications and tools (1) is managed by an orchestration server (2) and accessed by users via the Spekl command line tools (3).

libraries and packages (e.g., Swing or SWT), all of which may require specifications to be present in order to verify the program the user is writing.

Some verification tools handle this problem by distributing the full set of specifications available for their given tool when the user performs an installation. However, this approach is problematic for a number of reasons. First, programs and libraries evolve over time. Take for example the recent flaw found in Java's implementation of the TimSort algorithm. Though the essential behavior of the sorting algorithm remained the same (it was a sorting algorithm and put items in order), the *specification* of the method did in fact change [8].

In order for the user of a program verification tool such as OpenJML to get the update to that specification they would have to download a new version of the verification tool. In the Spekl model, a user can subscribe to specific versions of named specification libraries that are automatically kept up to date by the Spekl tool. This helps users to verify their programs against the latest specifications. This is helpful not only in the case that a refined specification is discovered; large code bases (for example, the Android API) may have partial or incomplete specifications. Spekl's automatic update mechanism allows program authors to gain access to revised specifications earlier, as they become available.

3.3 Specification Layering and Authoring

Writing a specification that will be applicable to every user is a challenging (if not impossible) task. For example, suppose a user is using a taint analysis tool that classifies all input from external devices as tainted and therefore should be classified as a defect in any program that allows its input to come from external devices. Although this may be desirable in some scenarios, the authors of the specification, by being overly specific, risk making the specification unusable for other purposes. If the user of the specification would like to specify that input from some trusted external device (say, a crytographic USB key) is not tainted the user will be unable to specify this behavior without finding ways of subverting the existing specification.

Unfortunately, the way in which a user will typically perform this task is to modify the specifications directly to override the specified behavior. However, this has the disadvantage that the specifications are now orphaned from the revision history from which they are derived. Edits to the specification library will become the technical debt of the project author and any subsequent updates to the base specifications will require the author to manually merge in changes from the upstream package. Even if the user does manage to succeed in modifying the offending specification, they will have now *diverged* from the original specification; any time the user updates the tool they are using, they must remember to update the specification they have modified. To complicate matters, if the specification they have modified has also changed in the tool package, then they must manually merge their changes with the changes distributed by the tool package. The manual effort required by this process makes it brittle and error-prone.

Simply being able to access predefined software specifications is an important part of the specification adoption problem. However, an important and under-addressed aspect of the problem is what to do when the existing reference specifications do not meet the needs of the user. We refer to specifications that have been modified for use in a new context as *child* specifications. While this topic will be covered in detail in Sect. 6, the final feature unique to Spekl is the ability to allow users to not only modify but subsequently share specifications. A core part of our approach, as indicated in Fig. 2, is that those child specifications are then optionally published to an external repository where they may be consumed by other users. New specifications in turn may either be freshly authored (using no existing specification library as a basis) or written by adapting existing specifications.

Since the focus of Spekl is specification authoring and reuse, in Sect. 6 we provide additional details the features of Spekl related to specification authoring and usage. Finally, in Sects. 6.2 and 6.3 we provide details about Spekl's specification layering and extension features.

Now that we have given an overview of the Spekl system, we turn our attention to detailing the various features of Spekl. In the next we will discuss in detail the facilities that Spekl provides for verification tool management.

4 Verification Tool Management

The first hurdle in getting users to verify their software is installing the verification tools. This presents a problem not only in software engineering contexts [6,13], but also in teaching scenarios, where tools must be installed onto the computers of students. To that end, the first problem Spekl solves is the problem of distributing program verification software. This section discusses how Spekl allows tool creators to author and publish tools to the Spekl repository as well how it helps users install and run these tools.

4.1 Creating New Tools

The first step to creating a new tool in Spekl is to use the `spm init` command. This command will prompt the tool author for basic metadata concerning their new tool. As a first step, Spekl will prompt the tool author for their GitHub username, under which they will publish the tool. The `init` command is executed as follows:

```
~/my-tool$ spm init tool
```

Spekl prompts the user for the tool's package name and version. If the current working directory is already a spekl-managed directory (meaning it contains a `spekl.yml` file), then a new tool is created at `.spm/<name>-<version>`. Otherwise, a new `package.yml` file is created in the current working directory. In addition to creating a new `package.yml` file, the `spm init` command also registers the tool's package name in the central Spekl repository, reserving that tool name. Spekl package names are assumed to be universally unique, thus attempting to use a preexisting package name is not allowed.

4.2 The Spekl Package Format

The `package.yml` that was created with the `spm init` command is the basis for configuring Spekl in the context of both tools and specifications. In this section we will be examining the package format as it pertains to tool authoring tasks. As a guiding example throughout this section will be referring to the example in Fig. 3, which is the `package.yml` file for the OpenJML tool.

Package Metadata. The first section of the `package.yml` file is concerned with author metadata. In this section the tool author should indicate the name of the package, the initial version number, and the kind of package it is. The `kind` field may be one of `tool` or `spec`. As we will see later in the section on publishing packages, the `author` field of the `package.yml` file is important; an author may specify one or more authors in this section and this information will be used during the authentication process during publishing later in the package lifecycle.

```
name : openjml-esc
version : 1.7.3.20150406-3
kind : tool
description: The Extended Static Checker for OpenJML.

author:
  - name: John L. Singleton
    email: jsinglet@gmail.com

depends:
  - one-of:
    - package: z3
      version: ">= 4.3.0 && < 4.3.1"
      platform: all
    - package: why3
      version: "> 0.80"
      platform: all
  - package : openjml
    version : ">= 1.7.3 && < 1.8"
    platform : all

assets:
    - asset : MAIN
      name : openjml-dist
      url : http://jmlspecs.sourceforge.net/openjml.tar.gz
      platform: all

assumes:
   - cmd: java --version
     matches: java version "1.7.*
     message: "Requires Java, version 1.7."

install:
  - cmd: tar -zxvf MAIN
    description: Unpacking the archive...
    platform: all
  - cmd: touch openjml.properties
```

Fig. 3. The `package.yml` file for the OpenJML-ESC package.

Expressing Package Dependencies. Many verification tools depend on the presence of other external tools. However it is not always possible for tool authors to distribute all of the binaries they may require in order to function. For example, many program verification tools require SMT solvers to be installed. Since there are many viable SMT solvers (some of which only work on certain platforms), rather than distributing all possible SMT solvers, tool authors might rely on the user to install the SMT tool of their choice. Spekl automates this process by allowing tool authors to declaratively express the external packages their tool depends on.

In Fig. 3, the OpenJML package expresses two types of dependencies with two different nodes under the `depends` keyword. This feature supports two types of dependencies. The first kind is a required dependency, which must be satisfied prior to installation. The second kind is a `one-of` dependency. As in the example with the SMT solvers, a user may wish to install one of several different possible options. Specifying a list of dependencies in the `one-of` node of the `package.yml` file instructs Spekl to prompt the user for a choice during installation. If a given tool should only be installed on a certain operating system type, a tool author my indicate one of `windows`, `osx`, `linux`, `unix`, or `all` to indicate which operating systems require the dependency being expressed.

Package Assets. A given tool may be the amalgamation of several different software artifacts installed into a single host system. In Spekl, a tool's *assets* are the individual bits of software that make up a tool. To specify an asset a user must create a tuple of `(asset, url, platform)` under the `assets` node in `package.yml`. The `asset` attribute creates a new binding to be used later in the installation process. In the example in Listing 3, the `MAIN` name is bound to the asset that will be eventually downloaded from the location at `url`. This name may be used symbolically in the commands found later in the `install` section of `package.yml`.

Establishing Environmental Assumptions. In addition to the built-in dependency management system, Spekl is also able to verify environmental assumptions that must be true in order for package installation to be successful. In Fig. 3 OpenJML assumes that the binary in the current user's path for `java` must be version 1.7. Note that environmental assumptions are different from dependencies in that they pertain to the state of the user's computer, where as dependencies pertain to the internal state of a user's set of installed packages.

To specify an environmental assumption, a user may add a node under the `assumes` node of the `package.yml` file by specifying a tuple of the form `(cmd, matches, message)`. The condition for proceeding is the logical conjunction after executing `cmd`, comparing the output with the regular expression contained in `matches`, and if the regular expression does not match the output of `cmd`, the message specified by `message` is displayed.

Specifying Installation Commands. The last section of the `package.yml` file is concerned with the actual commands necessary to install the tool onto the users' system. Unlike system-wide package installation tools like the APT package manager, Spekl installs tools and specifications on a project basis. This feature enables users to use different versions of verification tools (and different versions of their dependencies) on different projects without causing conflicts arising from installing conflicting tools (and possibly versions of tools) on the same system.[2]

To achieve this, Spekl maintains an installation database under the `.spm` directory within a Spekl project. For example, consider the directory listing shown in Fig. 4. In this listing we see that two packages are installed: openjml and z3.

To specify the installation commands for a package the user must specify a set of tuples of the form `(cmd, platform)`. The commands that apply to the destination platform will be executed in sequence with the following phases:

[2] The disadvantage of Spekl's technique is the use of more disk space than sharing a single installation of each tool.

```
/my-project
 ├── README
 ├── Main.java
 ├── Util.java
 ├── Tree.java
 ├── build.xml
 └── .spm
      ├── openjml-1.1.1
      │    ├── package.yml
      │    ├── jmlruntime.jar
      │    ├── jmlspecs.jar
      │    └── README
      └── z3-4.3.2
           ├── package.yml
           ├── bin
           ├── doc
           └── README
```

Fig. 4. An example directory layout for a Spekl project. Note that packages are wholly installed under the `.spm` directory.

Dependency Resolution. Prior to starting the installation of the current tool, the `depends` section of the tool will be examined and any tools that must be installed will be installed.

Database Preparation. After dependencies are resolved, Spekl will create a directory hierarchy to support the installation of this package. The installation directory will be located in the `.spm/<package>-<version>` directory.

Asset Acquisition. Prior to starting the installation commands specified in this section, each asset listed in the `assets` section will be downloaded to the local computer and placed in the database directory created in the previous phase.

Binding Substitution. The `assets` section of the `package.yml` file also establishes bindings for later use in the installation process. In the listing in Fig. 3, the `assets` section establishes a binding between `MAIN` and the eventual destination of the downloaded archive. This is useful since the tool author can (as shown in Fig. 3) later use the `MAIN` binding in a command that unpacks the downloaded archive. These bindings are valid throughout the `install` section of the tool's `package.yml`.

Installation Command Execution. The command set for the current platform is then extracted and executed in sequence. The sequence the commands are executed in is the sequence that they are specified in the file and in order for an installation to succeed all of the commands must succeed. The failure of any command causes the current installation to halt. Each command should be written to assume it is being executed in the installation directory for the package.

4.3 The Check Definition Language

Spekl's tool packages serve two purposes. The first is to provide a set of *checks* that the user may run on their software. The second is to provide resources (such as SMT solvers) for other checks and packages. In this section we focus on one of the most central aspects of tool authoring: defining the checks the tools are capable of running.

```
(ns spekl-package-manager.check
 (:require [clojure.java.io :as io]
           [clojure.tools.logging :as log]
           [spekl-package-manager.util :as util]))

(def report-file "findbugs-report.html")
(def report-file-xml "findbugs-report.xml")

(defn open-report [file]
  (log/info "Opening report...")
  (let [open-command (util/get-open-command file)]
      (apply run open-command)

      ))

(defcheck default
  (log/info  "Running findbugs... report will open after running...")
  (let [result  (run-no-output "java" "-jar" "${findbugs-3.0.1/lib/findbugs.jar}"
       "-textui" "-html" *project-files*)]
    (if (= 1 (result :exit))
      (println (result :out))
      (do
        (spit report-file (result :out))
        (open-report report-file)))))


(defcheck xml
  (log/info  "Running findbugs [XML]... report will be available at"
       report-file-xml "after running")
  (let [result  (run-no-output "java" "-jar" "${findbugs-3.0.1/lib/findbugs.jar}"
       "-textui" "-xml" *project-files*)]
    (if (= 1 (result :exit))
      (println (result :out))
      (spit report-file-xml (result :out)))))
```

Fig. 5. An example Spekl check file from the FindBugs package. This check file defines two checks: a default check that outputs FindBugs reports in HTML format and a check named "xml" that outputs FindBugs reports in XML format.

A Spekl tool may define any number of checks within a single package that may in turn be run by a Spekl user. To specify that a package should export checks, a tool author should create a `check.clj` file in the root directory of the tool package. The check defined in Fig. 5 shows the details of the checks available in the FindBugs [1] package. We describe the most pertinent portions of this check below.

As indicted by the `.clj` file extension, the host language of all Spekl checks is the programming language Clojure, a modern Lisp dialect that targets the Java Virtual Machine [9]. As such, tool authors have the full capacities of the Clojure language at their disposal while writing checks. However, to make this process easier, Spekl defines a special `defcheck` form that simplifies many tasks. The features of this form are described next.

The `defcheck` Form. To declare a new check, a tool author must use the `defcheck` form as shown on lines 16 and 26 of Fig. 5. These lines define the checks `default` and `xml`, respectively. Once defined, they can be referenced in a client's `spekl.yml` file, using the `check` configuration element. The `default` keyword has a special meaning in a Spekl check; although Spekl check files may define as many checks as they wish, all check files must at a minimum define the `default` check. The only other requirement of a check file is that they reside in the `spekl-package-manager.check` namespace, which is accomplished by the `ns` form on line 1.

The facilities that `defcheck` provides come in two forms: special variables and special forms. The special variables are:

check-configuration This variable contains the configuration information for the current check, as read from client's `spekl.yml` file. This feature is especially useful for tool authors that require additional configuration parameters to be present in order to successfully execute a verification tool.

project-files Prior to running a check, Spekl expands the entries given in the `paths` configuration element of a check (see Fig. 6). This element may contain a list of literal paths, relative paths, or shell globs that specify collections of files. These paths are resolved on the local file system and provided to the check in the **project-files** variable. Since tools often run on collections of files, this feature allows the tool to remain decoupled from the project that is using it. In the example in Fig. 5, the **project-files** variable is used as an argument to the FindBugs checker on lines 18 and 28.

specs This variable is similar to the **project-files** variable with a few important differences. In Spekl, specifications reside in packages that are stored in Spekl repositories. Since specifications may be found by version as well as name it is not possible to know statically where all the required specifications for a project may be located. Prior to running a check, Spekl resolves all of these specification paths and provides them to the check environment in the form of the *specs* variable (see Fig. 6 for an example of how this variable is configured).

As mentioned in Sect. 4.3, Spekl checks are hosted in the Clojure language. This gives users access to a wide array of Clojure-specific functions as well as any code that runs on the JVM. In addition to this functionality, Spekl defines an asset resolution functionality that can be seen on lines 18 and 28, in Fig. 5. The `run-no-output` form (and its complement, `run`), takes a variable number of string arguments that are then invoked on the host system. When one of the strings is of the form "${pkg:asset}", this causes Spekl to attempt to resolve the asset described between the curly braces. The token to the left of the colon is the *canonical name* of the package in which one expects the asset to reside. Since multiple versions of the same package may be installed on the users's system, Spekl automatically inspects the dependencies declared by the current tool and supplies only the package that complies with the restrictions set forth in the tool's `package.yml` file. If the left side of the colon is omitted, Spekl will attempt to resolve the resource in the currently executing tool's package. A failure to resolve any asset results in the termination of the current check.

4.4 Publishing Tools to the Spekl Repository

One of the challenges of creating a robust repository of software is simultaneously unglamorous and exceedingly hard: safely storing published packages. Rather than focus on the problem of building a computing infrastructure to support the storage of a software repository capable of handling possibly millions of users, Spekl has taken the approach of leveraging an existing service to handle the storage problem: GitHub. This comes with a number of advantages. First, using the free hosting capabilities of GitHub allow Spekl to remain both free and scalable. Second, since Spekl's packages are stored in Git repositories, it allows package authors fine-grained control over collaboration and workflow [15,16].

Once a package has been authored, Spekl allows a user to publish their tool directly from the command line interface. To initiate the publishing process the user should execute the `publish` command of Spekl as shown in the following example.

```
~/my-tool$ spm publish
[spm] Performing pre-publishing sanity check...
[spm] Current directory is a package directory
[spm] Creating new version: 0.0.1
[spm] Publishing changes to the repository
```

As mentioned in the previous section, all Spekl packages are publicly stored on GitHub. To that end, if an author wishes to directly access the repository (for example, to add a collaborator) they may access it immediately after publishing at the URL: https://github.com/Spekl/<package-name>. Note that since Spekl allows direct access to its repositories via Git, problems may arise if users maliciously edit repositories (for example, deleting them). Such actions could violate the invariants that Spekl requires to function. For the purposes of this work we assume users act in good faith and do not work to subvert Spekl.

4.5 Installing Tools from the Spekl Repository

Tool installation is the primary method by which end users of the Spekl system are expected to utilize verification tools. Any Spekl tool may be directly installed without configuring the project via the `spekl.yml` project file (covered in the next item). To do this a user may execute `spm` as follows:

```
~/my-project$ spm install openjml
```

This command begins an installation of the OpenJML tool for the current project. Note that this command takes an extra optional parameter `version`, which, if specified, tells Spekl which version of OpenJML to install.

While a tool may be directly installed as shown above, most users will install tools via the `spekl.yml` file. This will be covered in more detail in the next section, but a tool may be flagged for installation by being listed as a required *check* for a project.

Since Spekl is a centralized repository, the `spm` command takes advantage of this fact and provides users with a way to locate packages to install. The command that shows a user all available packages can be see in the following listing:

```
~/my-project$ spm list tools
```

This command will print a listing of the newest versions of tools available on Spekl along with the version numbers and descriptions of each tool.

5 Consuming Specifications

Once users are able to install verification tools they will need to combine them with specifications. The following sections are concerned with the problem of consuming specifications. We will begin with a description of how to create new Spekl projects.

In Spekl, the normal use case for end users of Spekl is to consume packages (tools and specifications) for use in their own projects. Similar to tool creation in the previous section, the way a user may indicate an existing project should be used with Spekl is to use the `init` command of `spm` as shown in the following example.

```
~/my-project$ spm init
```

This command prompts the user for some basic metadata about their project and creates a new `spekl.yml` file in the directory that the `spm` command was executed in.

5.1 The Spekl Specification Project Format

As discussed in the previous section, the basis for consuming specifications and tools happens at the project level and is configured via the `spekl.yml` file. This section will look at the `spekl.yml` file in detail.

Package Metadata. Unlike the `package.yml` metadata, the metadata section of the `spekl.yml` file does not contain information that will be made publicly available. Often a project will not be published (e.g., if it is internal to a company). As such the metadata section may be customized with whatever information is useful for describing a project.

```
name : My Project
project-id : my.test.project
version : 0.1

checks :
  - name : openjml-rac
    language : java
    paths : [src/**.java]

    tool:
      name : openjml-rac
      version : 1.1.12
      pre_check :
      post_check:

    specs:
      - java-core: 1.7
```

Fig. 6. An example `spekl.yml` project file.

Specification and Tool Configuration. The bulk of work editing a `spekl.yml` file is concerned with adding checks and specifications to the current project. This is done by manually editing the `checks` section of the `spekl.yml` file and will be the topic of this section.

One key feature of Spekl is its support for multiple checkers within a single project. This is especially useful for large, complex code-bases where one verification tool may not be enough to handle all verification concerns. For example, certain critical portions of the code base may need to be checked with static checking, which is more precise, but more demanding in terms of specifications, whereas it may be sufficient to check the remainder of the code base with runtime assertion checking.

To configure a checker, a user adds a node below the `checks` node of the `spekl.yml` file. In Fig. 6 we can see the OpenJML Runtime Assertion Checker configured for a project. A few items in this configuration are of note.

name. The name attribute of a check is used to identify the check to the user at runtime and any suitable string may be used.

language. Certain checkers may take advantage of knowing the input language of a project ahead of time. This field is optional and should only be specified if a tool needs it.

paths. The *paths* element of a check is a critical component of its configuration and is the method by which the verification tool will find source files to check. The paths element is a comma-separated list of shell globs that should contain all of the files expected to be passed to the backend verification tool. In the example, the double star pattern (`**`) means to match any directory within `src/` before matching the file pattern.

The next section of the check configuration pertains to selecting a tool and version to use. When `spm check` is run it will consult this section first to ensure that all dependencies are fulfilled before running the check. The parameters of this section are as follows:

name. The name of the tool to use as reported by the `spm` list command.

version. The version of the tool to require. In addition to numerical version numbers, this attribute also supports symbolic version number strings. For example to specify that version greater than 4.3 is required, a user can supply the version string ">4.3". Both conjunction and disjunction are supported and version strings such as ">4.3 && <5.0" may be supplied. The operators supported are the binary comparison operators: >, <, ==, <=, and >=.

pre_check/post_check. Though not normally needed, the pre_check and post_check configuration parameters are hooks to allow a user to run custom actions before and after a check. For this parameter, a user may specify a shell command to execute. No validation on this input is performed but a failing pre_check (a command returning a non-zero exit code) will cause the check's execution to halt.

5.2 Running Verification Tools

Once the `spekl.yml` file is configured, a user may run their verification tools by using the `spm` command from within the directory where the `spekl.yml` file is located. In this section we provide details on this process.

The general command a user may use for checking their programs is the `spm check` command as shown in the following example:

```
~/my-project$ spm check
```

Executing the `spm check` command in this way causes all of the configured checks in the `spekl.yml` file to be executed in sequence. If the user has multiple checks configured and only wishes to run one of the checks, they may refine the behavior of the `spm check` command by naming the check to run:

```
~/my-project$ spm check openjml-rac
```

As shown in Fig. 6, the configured check name is `openjml-rac`. Specifying `openjml-rac` as an argument to `spm check` indicates that the `spm` tool should only run the that specific check and no other.

6 Specification Authoring Features

Similar to the facilities for tool authoring, Spekl also provides support for specification authoring. Specifically, Spekl provides support for two modes of specification authoring: *specification creation* and *specification extension.*

In *specification creation*, a specification author creates a new specification for a body of code from scratch. The code being specified need not be the work of the specification author.

In *specification extension*, as we shall see in the following sections, a specification author creates a new specification based on an existing specification within the Spekl repository. The reason an author may do this is two-fold. First, the author may need to change the meaning of an existing specification to fit their needs. The second case is that the author may need to *add missing specifications* to the specification library to cover portions of the codebase not handled by the original specification author.

6.1 Creating New Specifications

To create a new specification from scratch, a specification author uses the spm init command as shown previously. The command is invoked as follows:

```
~/my-project$ spm init spec
```

After prompting the user for some basic metadata about the specification, the spm tool will create a package.yml file either in the current directory or in the .spm directory, depending on how the command was invoked before. Much of the package.yml file is identical to the one used for tools. The main difference is that for specifications the sections for assets, installation steps, and dependencies do not apply. Rather, the specifications must be wholly self-contained within the package itself. Similar to tools, completed specifications that are either extended or created from scratch may be published to the Spekl repository with the spm publish command.

6.2 Layering Specifications

In Sect. 2 we presented several examples where adapting a specification was not possible. In this case, rather than *modifying* a specification, what is needed is the ability to *extend* the specification in such a way that it can override the specification from which it is derived. This is a core idea behind how Spekl allows specification writers to use and extend specifications. In this section we give an example of this usage.

To begin, a user wishing to extend a specification should execute the extend command of spm with the name of the package specification they wish to extend as an argument. In our example we will extend jml-core-java-7 which is the set of specifications in JML for Java 7.

```
~/my-project$ spm extend jml-core-java-7
```

Executing this command creates a new specification project in the directory from which it was invoked. Let us take a look at the package.yml file that was created. The file is shown in Listing 1.3. As can be seen, this command adds the extends keyword to the package.yml file. The effect of this keyword has is to introduce a link between the revision history of the package it extends and the history of the new package. This keyword instructs Spekl that it should attempt to *merge* the revision histories of the parent and derived specifications before checking a specification. Any additions or modifications that have been made to the parent specification will automatically be made available in the derived specification. This *derived specification*, which may contain zero or more modifications, may then be published to the Spekl repository if desired and consumed by other authors who may find the modifications of the specification useful.

Listing 1.3. An excerpt from the `package.yml` file generated by the `spm extend` command

```
name : my-jml-java-7
version: 0.0.1
kind : tool
extends: jml-core-java-7
```

6.3 How Spekl Manages Hierarchies

One idea that Spekl introduces is the concept of *specification hierarchies*. In this section we describe specification hierarchies in detail and provide details on how specifications propagate throughout the hierarchy.

To begin, we consider a revision history of some specification S, $\mathscr{H}$, where $\mathscr{H}$ is a temporally ordered list of *revision strings* that specify the differences between two adjacent revisions in a history. An individual element of $\mathscr{H}$ is some element δ_n which is computed by taking the difference between the $n-1^{th}$ revision of S and the n^{th} revision of S. The initial difference in $\mathscr{H}$, δ_0, is defined as the empty revision string. Given an initial specification S and a revision to S, S', we write the difference between S and S' as $S - S'$.

A *specification hierarchy* is defined as the semilattice of pairs $(S, \mathscr{H})$, ordered by the refines relation, $\sqsupseteq$, that is:

$$(S', \mathscr{H}') \sqsupseteq (S, \mathscr{H}) \iff S' <: S \wedge (\exists \delta \in \mathscr{H}' :: \delta \in \mathscr{H}) \tag{1}$$

The meaning of the $S' <: S$ is that S' is declared to extend S. In Eq. (3) and below we refer to a specification hierarchy by the more compact notation, $\overrightarrow{SH}$, which is expanded more explicitly in (2).

$$\overrightarrow{SH} = \{(S_\bot, \mathscr{H}_\bot), \ldots, (S_\top, \mathscr{H}_\top)\} \tag{2}$$

$$\mathbf{extends}(S, \overrightarrow{SH}) = \left\{ \begin{array}{ll} nil & \text{if } S = S_\top \\ (s, h) \in \overrightarrow{SH} \;:\; S <: s & \text{otherwise.} \end{array} \right\} \tag{3}$$

In Eq. (2) the symbols $\top$ and $\bot$ refer to the specification at the top of the hierarchy (extended by every specification in the hierarchy) and the specification at the bottom of the hierarchy. Both $\top$ and $\bot$ must be uniquely defined and exist.

Note that in the case of the $\bot$ specification this does not imply that no specification extends it, but only that for a particular context there is no specification that extends it.

Specification hierarchies are useful in that they allow new specifications to be based off of existing specifications, thus benefiting from the pre-existing work in creating the base specification. We call specifications based off of existing specifications *child specifications*. Conversely, the direct specification on which a child specification is based is called the *parent specification*. We refer to the collection of all parent specifications (that is, the ancestor chain) as the *upstream specifications*.

What is unique about Spekl's specification hierarchies is that they allow a given specification to be modified while maintaining a historical connection to the revision history of any upstream specification that it extends. Any changes made to the upstream specification are automatically propagated down the chain. We refer to the current version of a given specification with all upstream changes applied as the *effective specification* of S.

Suppose now that a specification S' is created based on the specification S. The user extending the specification makes changes to S'. Meanwhile, the original author of S makes changes to it as well. These changes may include additions, deletions, or refinements of specifications. To complicate matters further, the upstream specification itself may in turn be based off another specification. How can we provide an appropriate effective specification?

To answer this question, Spekl defines two orthogonal concepts: specification refreshes and specification merges.

A *specification refresh* is an operation invoked by Spekl during specification checking that inspects the current specification, determines if there have been changes in the specification extension chain and applies those changes if needed in order to arrive at a new effective specification.

We provide a definition of the refresh operation in Eq. (4).

$$\begin{aligned}&\mathbf{refresh}((S,\mathscr{H}),\overrightarrow{SH})\\ &\quad=\left\{\begin{array}{ll}\mathscr{H}, & \text{if } \mathbf{extends}(S,\overrightarrow{SH})=nil\\ \mathbf{let}\ (S',\mathscr{H}')=\mathbf{extends}(S,\overrightarrow{SH}) & \text{otherwise.}\\ \quad\mathbf{in\ refresh}((S',\mathscr{H}\lhd\mathscr{H}'),\overrightarrow{SH}) & \end{array}\right\}\end{aligned} \tag{4}$$

The second concept, *specification merges*, is introduced by the presence of the operator $\lhd$ in Eq. (4) — the *merge* operator. The merge operation in Spekl ($\lhd$) is in fact the same three-way merge operation used by the version control system Git, which is based on the finding the longest common subsequence between two revisions of a file. It is different than the more simplistic two-way merge in that it takes into account the information about common ancestors in a revision history [14]. Improving the usefulness of the merge operation is a task relegated to future work for Spekl.

However, in order to understand how this merge operation impacts specifications, let us consider the possibilities of merging two arbitrary specifications and histories $(S,\mathscr{H})$ and $(S',\mathscr{H}')$:

Case 1 $(\forall\delta\in\mathscr{H}'{::}\delta\notin\mathscr{H})$ **Action:** No specification merge is possible since S' was not derived from S. All of the remaining cases assume that case 1 does not hold.

Case 2 $(\mathscr{H}\subseteq\mathscr{H}')$ **Action:** No action is needed since all of the history for S is already subsumed by the history of S'.

Case 3 $(\exists\delta\in\mathscr{H}{::}\delta\notin\mathscr{H}')$ **Action:** There has been at least one change to S that has not been applied to S'. These changes will be merged into S' by using the three-way merge algorithm described earlier. Note that this condition assumes that $(S', \mathscr{H}')$ refines $(S, \mathscr{H})$, i.e., the ordering relation $\sqsupseteq$ from Eq. (1) holds.

In Spekl, specification refreshes are invoked in two ways. The first way is automatically during the `spm check` command. During the execution of `spm check`, Spekl automatically collects the required specifications for a check and checks the remote repositories of the specifications to determine if they need to be refreshed. Additionally, if during specification development, an author wishes to synchronize their work with the upstream specifications, they may invoke the `spm refresh` command to manually trigger a refresh. The effective specification then becomes part of the permanent revision history of the child specification.

7 Related Work

The main related work on specification management (research questions 1 and 2) is that of source code control systems, such as Git. The reasons why such source code control systems are inadequate for using and extending specifications were discussed in Sects. 2 and 6.2.

Although Spekl does aim to abstract the process of installing and using verification tools, it does not aim to be a general purpose package management system. Nevertheless, Spekl implements many of the functions that are required by a software package management system and is therefore comparable to previous work in that domain. Though Spekl concerns itself with the installation of software artifacts like package management systems such as YUM [20], APT [19], and Haskell's Cabal [5,11], the installation of artifacts is only one of Spekl's features whereas they are the main focus of typical package management systems.

Recent work by Tucker et al. [17] demonstrated the usefulness of integrating SMT solvers into the problem of solving dependency-related problems in software package repositories. This work was later extended by Ignatiev et al. [10] who demonstrated an improvement over the OPIUM system in the case of SMT solver timeouts. Spekl's current dependency resolution strategy does not use any of these sophisticated techniques but rather uses an optimistic approach in which a package installation proceeds until it cannot. For example, if a package expresses that it requires a version of tool T greater than 1.0 Spekl will install the newest version of T satisfying that requirement. In theory, it could be the case that installing an older version of T (greater than version 1.0) could satisfy some other dependency at a later stage in the dependency resolution stage. This sort of scenario could be detected by an SMT solver-based approach and is more comprehensive. We intend to implement this strategy in a later revision of Spekl.

Another issue that impacts Spekl is the problem of broken components in evolving software repositories. This issue is explored in depth by Vouillon [18] who investigates strategies for fixing "broken sets" in evolving software repositories. From the perspective of Spekl, the techniques explored in this work could be applied to Spekl's specification libraries. In the case of Spekl, the problem is not as straightforward as a constraint solving problem pertaining to versions of software packages, but rather tied to ensuring that software artifacts satisfy their specifications as the repository evolves. This idea is indeed promising, but has been relegated to future work for Spekl.

8 Conclusion

In this paper we introduced a new system called Spekl that is aimed at streamlining the process of specification authoring and usage. As we have shown, Spekl's design solves the three key subproblems of achieving wider adoption of program specification: tool installation/usage, specification consumption, and specification authoring and extension. Additionally, we saw how Spekl addresses the problem of adaptive reuse for specifications by providing tooling to support writing and reusing specifications. In addition to providing a description of the problems impacting specification reuse, in this paper we have provided a detailed discussion of the Spekl system from the perspective of tool authors, specification authors, and users of both tools and specifications.

In addition to being a proof of concept tool, Spekl is an effort to widen the adoption of specification, verification, and validation in general. The Spekl tools can be downloaded from http://www.spekl-project.org, and people wishing to contribute to the development of Spekl may join the project via http://www.github.com/Spekl.

Acknowledgments. The work of both authors was supported in part by NSF grants CCF0916350 and CNS1228695, and CCF1518897. The work of Leavens was also supported by NSF grants CCF0916715, CCF1017262. The authors would like to thank Rochelle Elva, Faraz Hussain, and Luke Myers for their comments on earlier drafts of this paper.

References

1. Ayewah, N., Hovemeyer, D., Morgenthaler, J., Penix, J., Pugh, W.: Using static analysis to find bugs. IEEE Softw. **25**(5), 22–29 (2008)
2. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy, SP 2008, pp. 339–353. IEEE Computer Society, Washington, DC, USA (2008). http://dx.doi.org/10.1109/SP.2008.20
3. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. (STTT) **7**(3), 212–232 (2005). http://link.springer.com/article/10.1007/s10009-004-0167-4
4. Carter, K., Foltzer, A., Hendrix, J., Huffman, B., Tomb, A.: SAW: the software analysis workbench. In: Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology, HILT 2013, pp. 15–18. ACM, New York, NY, USA (2013). http://doi.acm.org/10.1145/2527269.2527277

5. Coutts, D., Potoczny-Jones, I., Stewart, D.: Haskell: batteries included. In: Proceedings of the First ACM SIGPLAN Symposium on Haskell, Haskell 2008, pp. 125–126. ACM, New York, NY, USA (2008). http://doi.acm.org/10.1145/1411286.1411303
6. Craigen, D.: Formal methods adoption: what's working, what's not! In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.) Theoretical and Practical Aspects of SPIN Model Checking. Lecture Notes in Computer Science, vol. 1680, pp. 77–91. Springer, Heidelberg (1999). http://link.springer.com/chapter/10.1007/3-540-48234-2_6
7. Ernst, M.D., Just, R., Millstein, S., Dietl, W., Pernsteiner, S., Roesner, F., Koscher, K., Barros, P.B., Bhoraskar, R., Han, S., Vines, P., Wu, E.X.: Collaborative verification of information flow for a high-assurance app store. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1092–1104. ACM, New York, NY, USA (2014). http://doi.acm.org/10.1145/2660267.2660343
8. Gouw, S., Rot, J., Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() Is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification, pp. 273–289. Springer International Publishing (2015). http://link.springer.com/chapter/10.1007/978-3-319-21690-4_16
9. Hickey, R.: The clojure programming language. In: Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, pp. 1:1–1:1. ACM, New York, NY, USA (2008). http://doi.acm.org/10.1145/1408681.1408682
10. Ignatiev, A., Janota, M., Marques-Silva, J.: Towards efficient optimization in package management systems. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 745–755. ACM, New York, NY, USA (2014). http://doi.acm.org/10.1145/2568225.2568306
11. Jones, I.: The Haskell Cabal, a common architecture for building applications and libraries (2005)
12. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for java. SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006). http://doi.acm.org/10.1145/1127878.1127884
13. Luqi, Goguen, J.A.: Formal methods: promises and problems. IEEE Softw. **14**(1), 73–85 (1997). http://dx.doi.org/10.1109/52.566430
14. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. **28**(5), 449–462 (2002)
15. Ram, K.: Git can facilitate greater reproducibility and increased transparency in science. Source Code Biol. Med. **8**(1), 7 (2013). http://www.scfbm.org/content/8/1/7/abstract
16. Stanisic, L., Legrand, A., Danjean, V.: An effective Git and org-mode based workflow for reproducible research. SIGOPS Oper. Syst. Rev. **49**(1), 61–70 (2015). http://doi.acm.org/10.1145/2723872.2723881
17. Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: OPIUM: optimal package install/uninstall manager. In: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 178–188. IEEE Computer Society, Washington, DC, USA (2007). http://dx.doi.org/10.1109/ICSE.2007.59
18. Vouillon, J., Di Cosmo, R.: Broken sets in software repository evolution. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, pp. 412–421. IEEE Press, Piscataway, NJ, USA (2013). http://dl.acm.org/citation.cfm?id=2486788.2486843

19. Apt - Debian Wiki. https://wiki.debian.org/Apt, https://wiki.debian.org/Apt
20. Yum (Yellowdog Updater, Modified). https://www.phy.duke.edu/~rgb/General/yum_HOWTO/yum_HOWTO/, https://www.phy.duke.edu/~rgb/General/yum_HOWTO/yum_HOWTO/

FCL: A Formal Language for Writing Contracts

William M. Farmer(✉) and Qian Hu

McMaster University, Hamilton, ON, Canada
{wmfarmer,huq6}@mcmaster.ca

Abstract. A contract is an artifact that records an agreement made by the parties of the contract. Although contracts are considered to be legally binding and can be very complex, they are usually expressed in an informal language that does not have a precise semantics. As a result, it is often not clear what a contract is intended to say. This is particularly true for contracts, like financial derivatives, that express agreements that depend on certain things that can be observed over time such as actions taken of the parties, events that happen, and values (like a stock price) that fluctuate with respect to time. As the complexity of the world and human interaction grows, contracts are naturally becoming more complex. Continuing to write complex contracts in natural language is not sustainable if we want the contracts to be understandable and analyzable. A better approach is to write contracts in a formal language with a precise semantics. Contracts expressed in such a language have a mathematically precise meaning and can be manipulated by software. The formal language thus provides a basis for integrating formal methods into contracts. This paper outlines FCL, a formal language with a precise semantics for expressing general contracts that may depend on temporally based conditions. We present the syntax and semantics of FCL and give two detailed examples of contracts expressed in FCL. We also sketch a reasoning system for FCL. We argue that the language is more effective for writing and analyzing contracts than previously proposed formal contract languages.

Keywords: Contracts · Formal languages · Simple type theory · Observables · Deontic logic · Conditional agreements · Temporally based conditions

1 Introduction

A contract records, orally or in writing, a legally binding agreement between two or more parties [22]. Contracts come in many forms and are used for many purposes [6, 22]. Written contracts are artifacts that can be stored, analyzed, modified, and reused. As artifacts, contracts are usually expressed informally in a natural

This is a revised and extended version of [9]. This research was supported by NSERC.

© Springer International Publishing AG 2018
S.H. Rubin and T. Bouabana-Tebibel (eds.), *Quality Software Through Reuse and Integration*, Advances in Intelligent Systems and Computing 561,
DOI 10.1007/978-3-319-56157-8_9

language such as English. Since natural language does not have a precise semantics, it can be difficult to write complex ideas in a natural language in a clear and unambiguous way. Thus contracts that embody complex agreements can be very difficult to both write and understand when natural language is used.

The meaning of a contract — that is, what the agreement is — often depends on certain things that can be observed, called *observables*, such as actions taken by the parties of the contract, events that happen, and values (like a stock price) that fluctuate with respect to time. A contract of this kind is *dynamic*: the contract's meaning changes over the course of time. A dynamic contract contains temporally based conditions that trigger changes to the contract's meaning when the conditions become true. Since the structure of these conditions can be very complex, dynamic contracts can be very difficult to understand and analyze. For example, financial derivatives that *derive* their values from fluctuating underlying assets are dynamic contracts that are notorious for being difficult to value [4].

Contracts — in particular, dynamic contracts — are naturally becoming more complex as the complexity of the world and human interaction grows. Continuing to write complex contracts in natural language is not sustainable if we want the contracts to be understandable and analyzable. A better approach is to write contracts in a formal language with a precise semantics. Then a contract becomes a formal object that has a mathematically precise meaning and that can be manipulated by software. A *formal contract* of this kind can be written, analyzed, and manipulated in various ways with the help of sophisticated software tools.

This paper outlines FCL, a Formal Contract Language with a precise semantics for writing general contracts. In FCL, a contract is a set of components (definitions, agreements, and rules) that can refer to observables and can include conditions that depend on observables. The meanings of these components can change when the values of observables mentioned in them change, and new components can be added when conditions become true. Hence the state of a contract as a set of components can evolve over time in much the same way as the state of a computer program evolves over time.

The paper is organized as follows. Section 2 presents a simple example of a dynamic contract. Section 3 discusses what properties contracts have. An overview of FCL is given in Sect. 4, and the formal semantics of FCL is outlined in Sect. 5. Permissions and reparations are discussed in Sect. 6. Section 7 shows how the example from Sect. 2 can be expressed in FCL. A more complex example expressed in FCL is given in Sect. 8. A system for reasoning about contracts written in FCL is sketched in Sect. 9. How FCL is related to other formal and informal contract languages is summarized in Sect. 10. And the paper concludes with Sect. 11.

2 Example 1: An American Call Option

To illustrate the role of observables and conditions that depend on them in a dynamic contract, we will consider the following simple example.

Example 1. Consider an American call option for purchasing one share of a certain kind of stock on June 30, 2015 for \$5. The expiration date of the option is December 17, 2015 (and so the option may be exercised on any date from June 30, 2015 to December 17, 2015). The strike price of the option is \$80. The transaction of the sale of the stock must be finished within 30 days of payment.

An *American option* is a contract that gives the owner the right, but not the obligation, to buy or sell a specified asset at a specified price on or before a specified date [11]. This example describes the conditions that are required for the sale of one share of stock. It shows the role that observables and conditions commonly play in contracts. If a payment of \$5 on June 30, 2015 is made to the option seller to buy the option (first condition), the option contract will become effective. If the option buyer exercises the option by paying \$80 to the option seller on or before December 17, 2015 (the second condition), the option seller will transfer one share of the stock to the option buyer within 30 days after the option is exercised. The payments of \$5 and \$80 are both observables on which the first and second conditions respectively depend. The transference of the stock is also an observable.
This American call option can be deconstructed into three components:

1. **Condition 1:** The option buyer gives the option seller \$5 on June 30, 2015 to buy an American call option consisting of a conditional agreement composed of the following two components.
2. **Condition 2:** The option buyer chooses to exercise the option by paying \$80 to option seller on or before December 17, 2015.
3. **Agreement:** The option seller is obligated to transfer one share of stock to the option buyer no later than 30 days after the option is exercised.

The contract thus has the following form:

if **Condition 1**
then
if **Condition 2**
then **Agreement**

Both if-then parts of the contract are conditional agreements. The second conditional agreement consists of **Condition 2** and **Agreement**, and the first conditional consists of **Condition 1** and the second conditional agreement.

The *offer time* of the American call option is the time the option contract is offered by the option seller to possible buyers. At the offer time, the option contract is not a legally binding agreement. It becomes a legally binding agreement only when the option contract is purchased by the option buyer (i.e., the time when **Condition 1** is satisfied).

A conditional agreement, like either of the two in this example, can be viewed as a "rule" that generates an agreement depending on the values of certain observables. In general, different agreements are generated when observables have different values. Observables determine both the meaning of a contract and how the meaning of the contract evolves over time.

3 What is a Contract?

Before we present FCL, our formal language for writing contracts, we need to discuss what a contract is. A contract is an artifact with certain properties. There is not a clear consensus of which of these properties are necessary and which are optional. We favor the definition of a contract given by Brian Blum in [6, p. 2]. He says a contract must have each of the following properties:

– Is an oral or written agreement.
– Involves at least two parties.
– Includes at least one promise made by the parties.
– Establishes an exchange relationship between the parties.
– Is legally enforceable.

A contract is created only because the parties reach agreement on the terms of the contract. The *parties* are the people or entities that have mutually agreed to the contract and are bound by its terms and conditions. In the case of written agreements, the parties are typically identified as the people or entities that signed the agreement. For any contract to be valid, there must be at least two parties. Typically, one party makes an offer and the other party accepts it. In addition, to be valid a contract must involve the parties in an *exchange* of something of value such as services, goods, or a promise to perform some action. Note that the exchange of money is not necessary.

A contract involves a *promise* which Blum defines as an "undertaking to act or refrain from acting in a specified way at some future time" [6, p. 5]. We think of "undertaking to act" as the deontic notion of *obligation.* Similarly, we understand "refrain from acting" as the deontic notion of *prohibition.* Obligation and prohibition are concepts studied in deontic logic [17]. They have the distinctive characteristic of being violable. When a promise made in a contract is honored, we say the promise has been *satisfied.* If it has not been honored, we say it has been *violated.* A promise may be restricted by a temporal bound, that is, a period of time during which an obligation or prohibition is in force. For example, a tenant may be obliged to pay rent on the first day of each month.

We will use an expanded definition of a contract that includes "degenerate contracts" that would not be considered contracts according to Blum's definition but are convenient to include in the space of all possible contracts. For example, a contract is *void* if it violates the law [4]. Void contracts are not legally enforceable agreements, so by Blum's definition they are not genuine contracts. We will consider them to be contracts, but we will designate them as being degenerate. Similarly, we will consider an agreement between two parties that does not include a promise or establish an exchange relationship between the parties as a degenerate contract.

4 Overview of FCL

This section describes the main components of FCL and informally explains their purpose and meaning. The formal semantics of FCL is outlined in Sect. 5.

4.1 Underlying Logic

We will assume that the underlying logic of FCL is some version of simple type theory [8]. Simple type theory is a form of high-order logic with function types, quantification over functions, and function abstraction. The underlying logic must have the following base types:

1. Bool, a type consisting of the boolean values T (true) and F (false).
2. Time, a type consisting of the integers $\mathbb{Z}$. That is, we assume that time is represented as a discrete linearly ordered set of values such that each value has a predecessor and a successor. The values many denote any convenient measure of time such as days, hours, seconds, etc.
3. Event, a type of events. These can be actions performed by the parties of a contract as well as events that the parties have no control over.

The underlying logic must include the following constants:

1. true and false of type Bool.
2. $\mathsf{obs\text{-}event}$ of type $\mathsf{Time} \times \mathsf{Event} \rightarrow \mathsf{Bool}$.

true and false represent the truth values T and F, respectively. $\mathsf{obs\text{-}event}$ is used to express observations of events as described in Sect. 4.2. The underlying logic must also have the variable X_{time} of type Time. X_{time} is used to instantiate expressions with the current time of a contract.

An *expression* of FCL is any expression in the underlying logic of FCL. A *formula* of FCL is an expression of FCL of type Bool.

Building FCL on simple type theory gives FCL access to the high expressivity and reasoning power of simple type theory [8]. This means that FCL can be developed largely by utilizing the standard machinery of simple type theory without the need to develop new logical ideas.

4.2 Observables

An *observable* is something that has a variable value that can be observed at a particular time [20,21]. Let us look at a couple of examples. The temperature of a room is an observable. Its value at a given time t is the temperature measured in the room at t. An event is an observable whose value is either true or false. Its value at a given time t is true [false] if the event occurs [does not occur] at t.

An *observable* of FCL is the application of a constant f of type

$$\mathsf{Time} \times \alpha_1 \times \cdots \times \alpha_n \rightarrow \beta$$

where $n \geq 0$. Thus the value of the observable $f(t, a_1, \ldots, a_n)$ depends on time in the sense that it depends on the value of its first argument which is of type Time. The value of $f(t, a_1, \ldots, a_n)$ also depends on the parameters $a_1, \ldots, a_n$. An *observation* of FCL is an atomic formula of the form $o = v$ where o is an observable $f(t, a_1, \ldots, a_n)$ and v is a value in the output type of f. When the output type of f is Bool, $o = \mathsf{true}$ and $o = \mathsf{false}$ can be written as o and $\neg o$, respectively. An *observational statement* of FCL is a formula of the underlying logic of FCL constructed from observations using the machinery of the underlying logic — which includes propositional connectives, quantifiers, and the other usual machinery of simple type theory.

We will show how the two examples of observables mentioned above can be expressed in FCL. Let $\mathsf{obs\text{-}temp}$ be a constant of type $\mathsf{Time} \to \mathbb{Z}$. Then $\mathsf{obs\text{-}temp}(t) = a$ represents the observation that the temperature in a particular room is a at time t. $\mathsf{obs\text{-}event}(t, e)$ represents the observation that the event e occurs at time t.

4.3 Actions

An *action* is an event that can be performed by the parties of a contract. There are two sorts of entities involved in an action: *subjects* and *objects*. The former are the entities who perform the action, while the latter are the entities that are acted upon by the subjects. An *action* a of type Event is defined as a tuple of the form $(L, \alpha, \mathcal{S}, \mathcal{O})$ where L is the *label* of an action, α is the *act* of the action (i.e., the thing that is performed), $\mathcal{S}$ is the set of *subjects* of the action, and $\mathcal{O}$ is the set of *objects* of the action.

Contracts typically include actions that specify the transfer of resources (money, goods, services, and even pieces of information) between parties. The act of the action would be the transfer of resources from one party (the subject) to another party (the object). Notice that an action of this kind encodes both what is transferred and what parties are involved in the transference.

4.4 Constant Definitions

A *constant definition* of FCL is an expression of the form $c = e$ where c is a new constant or an application of a new constant and e is an expression that defines the value of c. Constant definitions are used, among other things, to define temporally based values.

4.5 Agreements

An *agreement* is a promise to do or not do a specific action. An *agreement* of FCL is an expression of either the form $\mathbb{O}(a, \mathcal{T})$ or the form $\mathbb{F}(a, \mathcal{T})$ where a is an action and $\mathcal{T}$ is a set of times. $\mathbb{O}(a, \mathcal{T})$ is called an *obligation*; it represents the promise that the action a will be observed at some time in $\mathcal{T}$. $\mathbb{F}(a, \mathcal{T})$ is called a *prohibition*; it represents the promise that the action a will not be observed at any time in $\mathcal{T}$. $\mathbb{O}(a, \mathcal{T})$ and $\mathbb{F}(a, \mathcal{T})$ are considered to be *duals* of each other. The operators $\mathbb{O}$ and $\mathbb{F}$ are inspired by the deontic operators for *obligation* and *prohibition* [17].

4.6 Rules

A *rule* R of FCL is inductively defined as an expression of the form

$$\varphi \mapsto \mathcal{B}$$

where φ is a formula of FCL and $\mathcal{B}$ is a set of constant definitions, agreements, and rules. We assume that each free variable occurring in a constant definition or an agreement in $\mathcal{B}$ also occurs in φ. We will see in Sect. 5 that, if φ is satisfied at time t, the members of $\mathcal{B}$ are added to the state of a contract at time $t + 1$. Thus a rule can dynamically change the meaning of a contract.

A rule of the form $\varphi \mapsto \{A\}$, where A is an agreement, represents a *conditional agreement.*

4.7 Contracts

A *contract* C of FCL is a pair $(t_{\text{offer}}, \mathcal{B})$ where t_{offer} is a time and $\mathcal{B}$ is a set of constant definitions, agreements, and rules. The *parties* of C are the parties mentioned in the agreements in $\mathcal{B}$. t_{offer} is the time the contract is offered to the parties.

As we will see in the next section, a contract has a state consisting of a set of constant definitions, agreements, and rules. The state evolves over time like the state of a program evolves over time. A contract is *fulfilled* when all the agreements in its state are satisfied and all the rules in its state are no longer applicable. A contract is *breached* when some agreement in its state is violated.

5 Formal Semantics of FCL

This section presents the highlights of the formal semantics of FCL.

5.1 Models

A *model* of FCL is a model of the underlying logic of FCL. Throughout this section let $\mathcal{M}$ be a model of FCL. Let $V^{\mathcal{M}}$ be the valuation function of $\mathcal{M}$ that assigns each (closed) expression of FCL a value in $\mathcal{M}$. In particular, $V^{\mathcal{M}}$ assigns each observable $f(t, a_1, \ldots, a_n)$ a value for all times t (and parameters $a_1, \ldots, a_n$). Thus a model includes the values for all observables over all time.

5.2 Agreements

Let $t \in \mathbb{Z}$ and $\bar{t}$ be some canonical expression whose value is t. The *value of an obligation* $\mathbb{O}(a,\mathcal{T})$ *in* $\mathcal{M}$ *at time* t is $V^{\mathcal{M}}(\varphi)$ where φ is the formula

$$\exists\, u : \mathsf{Time}\ .\ u \in \mathcal{T} \wedge u \leq \bar{t} \wedge \mathsf{obs\text{-}event}(u, a).$$

The *value of a prohibition* $\mathbb{F}(a, \mathcal{T})$ *in* $\mathcal{M}$ *at time* t is $V^{\mathcal{M}}(\psi)$ where ψ is the formula

$$\forall\, u : \mathsf{Time}\ .\ u \in \mathcal{T} \supset (u \leq \bar{t} \wedge \neg \mathsf{obs\text{-}event}(u, a)).$$

An agreement is *satisfied* in $\mathcal{M}$ at time t if its value in $\mathcal{M}$ at t is T. An agreement is *violated* in $\mathcal{M}$ at time t if the value of its dual in $\mathcal{M}$ at t is T. We will occasionally use an agreement $\mathbb{O}(a, \mathcal{T})$ or $\mathbb{F}(a, \mathcal{T})$ as a formula of FCL whose meaning is φ or ψ, respectively.

5.3 Rules

Let $R = \varphi \mapsto \mathcal{B}$ be a rule of FCL and $t \in \mathbb{Z}$. Define $\mathsf{sub}(\varphi, t)$ to be the set of substitutions σ that map the free variables in φ to appropriate expressions such that $\sigma(X_{\text{time}}) = \bar{t}$. The variable X_{time} is used to instantiate a rule with the current time of a contract. For any expression e and substitution $\sigma \in \mathsf{sub}(\varphi, t)$, let $e\sigma$ be the result of applying σ to each free variable in e if e is not a rule and to each free variable in e except X_{time} if e is a rule. Then define $\mathsf{new\text{-}items}(R, \mathcal{M}, t)$ to be

$$\{e\sigma \mid \sigma \in \mathsf{sub}(\varphi, t) \wedge V^{\mathcal{M}}(\varphi\sigma) = \text{T} \wedge e \in \mathcal{B}\}.$$

R is *active* in $\mathcal{M}$ at t if $V^{\mathcal{M}}(\varphi\sigma) = \text{T}$ for some $\sigma \in \mathsf{sub}(\varphi, t)$. R is *defunct* in $\mathcal{M}$ at t if $V^{\mathcal{M}}(\varphi\sigma) = \text{F}$ for all $u \geq t$ and all $\sigma \in \mathsf{sub}(\varphi, u)$. If R is defunct in $\mathcal{M}$ at t, then R is not active in $\mathcal{M}$ at u and $\mathsf{new\text{-}items}(R, \mathcal{M}, u) = \emptyset$ for all $u \geq t$.

5.4 Contracts

Let $C = (t_{\text{offer}}, \mathcal{B})$ be a contract. The *state* of C in $\mathcal{M}$ at time $t \geq t_{\text{offer}}$, written $\mathsf{state}(C, \mathcal{M}, t)$, is the set of constant definitions, agreements, and rules defined inductively as follows:

1. $\mathsf{state}(C, \mathcal{M}, t_{\text{offer}}) = \mathcal{B}$.
2. If $t \geq t_{\text{offer}}$, then $\mathsf{state}(C, \mathcal{M}, t + 1) =$

$$(\mathsf{state}(C, \mathcal{M}, t)) \cup \bigcup_{R \in \mathcal{B}} \mathsf{new\text{-}items}(R, \mathcal{M}', t)$$

   where $\mathcal{M}'$ is the smallest expansion of $\mathcal{M}$ such that $V^{\mathcal{M}}(\psi) = \text{T}$ for each constant definition $\psi \in \mathsf{state}(C, \mathcal{M}, t)$.

The model $\mathcal{M}'$ in clause 2 is called the *C-expansion of $\mathcal{M}$ at time t*. A *model of C at time t* is any C-expansion of a model of FCL at time t.

C is *fulfilled* in $\mathcal{M}$ at time $t \geq t_{\text{offer}}$ if every agreement in $\mathsf{state}(C, \mathcal{M}, t)$ is satisfied in the C-expansion of $\mathcal{M}$ at t and every rule in $\mathsf{state}(C, \mathcal{M}, t)$ is defunct in the C-expansion of $\mathcal{M}$ at t. C is *breached* in $\mathcal{M}$ at time $t \geq t_{\text{offer}}$ if there is an agreement in $\mathsf{state}(C, \mathcal{M}, t)$ that is violated in the C-expansion of $\mathcal{M}$ at t. C is *null* in $\mathcal{M}$ at time $t \geq t_{\text{offer}}$ if $\mathsf{state}(C, \mathcal{M}, t)$ contains no agreements and every rule in $\mathsf{state}(C, \mathcal{M}, t)$ is defunct in the C-expansion of $\mathcal{M}$ at t.

Notice that we are employing a very simple model of concurrency in our semantics for contracts: At each time t, all active rules are applied simultaneously and the resulting new components – constant definitions, agreements, and rules — are added to the state of the contract at the next point in time. There is no opportunity for the application of rules to interfere with each other. In particular, a component can never be removed from the state once it is added to it. It is possible, however, for a contract state to be produced that contains contradictions, but this would be caused by a flaw in the contract, not a flaw in the conceptual framework.

6 Two Additional Concepts

This section explains how permissions and reparations can be expressed in FCL.

6.1 Permissions

Agreements of the form $\mathbb{O}(a, \mathcal{T})$ and $\mathbb{F}(a, \mathcal{T})$ are used in FCL to represent promises in the form of obligations and prohibitions. Notice that agreements in FCL do not include expressions formed using an operator corresponding to the deontic operator for *permission*. Unlike an obligation or a prohibition, a permission is not a promise.

Some kinds of permissions can be expressed in FCL. For example, the permission to exercise a call option is expressed by adding a rule $\varphi \mapsto \mathcal{B}$ to the contract's state where the condition φ holds if it is observed that the buyer of the option exercises the option and $\mathcal{B}$ includes an obligation that the seller of option sells to the buyer the goods specified by the option. See Sect. 7 for details.

6.2 Reparations

A contract usually specifies actions to be taken in case of the violation of a part of the contract. A conditional obligation arising in response to a violated agreement is considered as a *reparational agreement*. We extend the example of a sale of a laser printer contract from [15, p. 5] to explain how a violation that arises in contract can be "repaired".

Example 2. The contract consists of five clauses:

1. Seller agrees to transfer and deliver to Buyer one laser printer within 22 days after an order is made.
2. Buyer agrees to accept the goods and to pay a total of \$200 for them according to the terms further set out below.
3. Buyer agrees to pay for the goods half upon receipt, with the remainder due within 30 days of delivery.
4. If Buyer fails to pay the second half within 30 days, an additional fine of 10% has to be paid within 14 days.

5. Upon receipt, Buyer has 14 days to return the goods to Seller in original, unopened packaging. Within 7 days thereafter, Seller has to repay the total amount to Buyer.

Note that clause 3 of this example is a primary obligation, saying that the buyer is obligated to pay the second half within 30 days of delivery. Clause 4 of is an example of *reparational obligation* in which an unfulfilled obligation can generate obligations to "repair" this violation. It says what the buyer is obligated to do if he or she violates the primary obligation.

Similar to the reparational obligation, a *reparational prohibition* is a conditional agreement arising in response to a violated prohibition. Both the reparational obligations and reparational prohibitions are reparational agreements. In FCL, a reparational agreement will be expressed as a rule that can create other agreements or rules in response to the violation of the primary agreement. To express the potential violations, we introduce $\mathsf{obs\text{-}event\text{-}during}(\mathcal{T}, e)$ to represent the observation that the event e occurred during the time period $\mathcal{T}$.

In FCL, a reparational obligation of $\mathbb{O}(a, \mathcal{T})$ is expressed as a rule of the form

$$\neg\mathsf{obs\text{-}event\text{-}during}(\mathcal{T}, a) \mapsto \mathcal{B}$$

where a is an action. $\neg\mathsf{obs\text{-}event\text{-}during}(\mathcal{T}, a)$ represents a potential violation of agreement $\mathbb{O}(a, \mathcal{T})$. If an obligation is satisfied, the rule to "repair" this obligation will never be active. Similarly, a reparational prohibition of $\mathbb{F}(a, \mathcal{T})$ is expressed as a rule of the form

$$\mathsf{obs\text{-}event\text{-}during}(\mathcal{T}, a) \mapsto \mathcal{B}$$

where a is an action. A rule that represents a reparational prohibition of $\mathbb{F}(a, \mathcal{T})$ will always be defunct if the agreement $\mathbb{F}(a, \mathcal{T})$ is satisfied.

Consider clause 4 of Example 2, the reparational obligation of the primary obligation given in clause 3 is the conditional agreement that, if the second half of payment has not been observed within 30 days of delivery, then the buyer has to pay an additional fine of 10% within 14 days. How this conditional agreement is expressed in FCL is shown in Sect. 8.

7 Example 1 Formalized: An American Call Option

We formalize here the American Call Option introduced in Sect. 2 as a contract C of FCL. C has two parties: a seller and a buyer. The unit of time is one day. Let the offer time t_{offer} of the contract, the time the seller offered the contract to the buyer, be some day before June 30, 2015. C is defined as the pair $(t_{\text{offer}}, \{D_1, D_2, R_1\})$ where:

$D_1 : t_{\text{buy}} = 0$ (June 30, 2015).
$D_2 : t_{\text{expire}} = 170$ (December 17, 2015).
R_1 is defined below.

C is constructed from two rules R_1 and R_2:

1. *Rule for Buying the Option:*
   $R_1 = \varphi_1 \mapsto \{R_2\}$ where:
   $\varphi_1 = \mathsf{obs\text{-}event}(t_{\text{buy}}, e_1)$.
   $e_1 = (\text{“Buy Option”}, \mathsf{transfer}(\$5), \{\mathsf{buyer}\}, \{\mathsf{seller}\})$.
   R_2 is defined below.

2. *Rule for Exercising the Option:*
   $R_2 = \varphi_2 \mapsto \{D_3, A\}$ where:
   $\varphi_2 = \mathsf{obs\text{-}event}(X_{\text{time}}, e_2) \wedge t_{\text{buy}} \leq X_{\text{time}} \leq t_{\text{expire}}$.
   $e_2 = (\text{“Exercise Option”}, \mathsf{transfer}(\$80), \{\mathsf{buyer}\}, \{\mathsf{seller}\})$.
   $D_3 : t_{\text{exercise}} = X_{\text{time}}$.
   $A = \mathbb{O}(e_3, [t_{\text{exercise}}, t_{\text{exercise}} + 30])$.
   $e_3 = (\text{“Transfer Stock”}, \mathsf{transfer}(\mathsf{stock}), \{\mathsf{seller}\}, \{\mathsf{buyer}\})$.

t_{buy}, t_{expire}, and t_{exercise} are new constants of type Time. $[t_{\text{exercise}}, t_{\text{exercise}} + 30]$ is the interval representing the set of times $\{t_{\text{exercise}}, t_{\text{exercise}} + 1, \ldots, t_{\text{exercise}} + 30\}$.

Each of the three events e_1, e_2, and e_3 are actions by one of the two parties. The three events are tied to the contract. For example, a more exact name for “Buy Option” would be “Buy Option Described by Contract C”. We assume that each of the three events can happen as most once. φ_1 asserts the option is bought on June 30, 2015, and φ_2 asserts the option is exercised at a time after the option is bought and before the option expires.

The *state* of C in a model $\mathcal{M}$ at time $t \geq t_{\text{offer}}$, written as $\mathsf{state}(C, \mathcal{M}, t)$, evolves over time as indicated in Fig. 1. How the state of C evolves depends on the observables specified by $\mathcal{M}$. In the figure, let u be the time that R_2 becomes active, i.e., when the buyer exercises the option. Let σ be the substitution that maps X_{time} to $\overline{u}$. Applying σ has the effect of replacing X_{time} with $\overline{u}$, whose value is the time u. $D_3\sigma$ is thus the equation $t_{\text{exercise}} = \overline{u}$, but $A\sigma$ is A since X_{time} does not occur in A.

8 Example 2 Formalized: A Sale of Goods Contract

We previously saw an encoding of the American Call Option in FCL. In this section we formalize the sale of the printer contract introduced in Sect. 6.2. Although this example contract is very simple, two points should be noticed. First, as illustrated in Sect. 6.2, this contract includes a reparational agreement that can used to repair a potential violation. Second, consider the total amount specified in clause 5. Taken literally, it would imply that the total amount the seller must repay to buyer in case of a return of the printer should be \$200 as stated in clause 2. Actually, this is certainly not the seller's intention. In fact, the total amount to be repaid should be the amount that the buyer has already paid the seller (which may not be the full \$200).

Now we formalize this contract in FCL to explain how we deal with the problems mentioned above. Let C be this contract expressed in FCL. C has two parties:

$$\begin{tikzcd}
\mathsf{state}(C, \mathcal{M}, t_{\text{offer}}) = \{D_1, D_2, R_1\} \arrow[r, "R_1 \text{ is defunct}"] \arrow[d, "R_1 \text{ is active}"] & \textbf{null} \\
\mathsf{state}(C, \mathcal{M}, t_{\text{buy}}+1) = \{D_1, D_2, R_1, R_2\} \arrow[r, "R_1, R_2 \text{ are defunct}"] \arrow[d, "R_2 \text{ is active}"] & \textbf{null} \\
\mathsf{state}(C, \mathcal{M}, t_{\text{exercise}}+1) = \{D_1, D_2, D_3\sigma, A\sigma, R_1, R_2\} \arrow[r, "A\sigma \text{ is violated}"] \arrow[d, "A\sigma \text{ is satisfied; } R_1, R_2 \text{ are defunct}"] & \textbf{breached} \\
\textbf{fulfilled} &
\end{tikzcd}$$

Fig. 1. Execution of the American Call Option C

a seller and a buyer. The unit of time is one day. C is defined as the pair $(t_{\text{offer}}, \{R_1\})$ where t_{offer} is the time when the seller offered the contract to the buyer.
C is constructed from the following nine rules:

1. *Rule for Ordering a Printer:*
   $R_1 = \varphi_1 \mapsto \{D_1, A_1, R_2\}$.
   $\varphi_1 = \mathsf{obs\text{-}event}(X_{\text{time}}, e_1) \wedge t_{\text{offer}} \leq X_{\text{time}}$.
   $e_1 = (\text{“Order Printer”}, \mathsf{transfer}(\mathsf{order}), \{\mathsf{buyer}\}, \{\mathsf{seller}\})$.
   $D_1 : t_{\text{order}} = X_{\text{time}}$.
   $A_1 = \mathbb{O}(e_2, [t_{\text{order}}, t_{\text{order}} + 22])$.
   $e_2 = (\text{“Deliver Printer”}, \mathsf{transfer}(\mathsf{printer}), \{\mathsf{seller}\}, \{\mathsf{buyer}\})$.
   R_2 is defined below.

2. *Rule for Delivering the Printer:*
   $R_2 = \varphi_2 \mapsto \{D_2, D_3, R_3, R_4, R_5, R_6\}$.
   $\varphi_2 = \mathsf{obs\text{-}event}(X_{\text{time}}, e_2) \wedge t_{\text{order}} \leq X_{\text{time}} \leq t_{\text{order}} + 22$.
   $D_2 : t_{\text{deliver}} = X_{\text{time}}$.
   $D_3 : \mathsf{obs\text{-}total\text{-}paid}(X_{\text{time}}) = 0$.
   R_3, R_4, R_5 and R_6 are defined below.

3. *Rule for Returning the Printer:*
   $R_3 = \varphi_3 \mapsto \{D_4, A_2\}$.
   $\varphi_3 = \mathsf{obs\text{-}event}(X_{\text{time}}, e_3) \wedge t_{\text{deliver}} \leq X_{\text{time}} \leq t_{\text{deliver}} + 14$.
   $e_3 = (\text{“Return Printer”}, \mathsf{transfer}(\mathsf{printer}), \{\mathsf{buyer}\}, \{\mathsf{seller}\})$.
   $D_4 : t_{\text{return}} = X_{\text{time}}$.
   $A_2 = \mathbb{O}(e_4(\mathsf{obs\text{-}total\text{-}paid}(X_{\text{time}})), [t_{\text{return}}, t_{\text{return}} + 7])$.
   $e_4 = \lambda X_{\text{total}}.(\text{“Return Payment”}, \mathsf{transfer}(X_{\text{total}}), \{\mathsf{seller}\}, \{\mathsf{buyer}\})$.

4. *Rules for Recording the Payment:*
   $R_4 = \varphi_4 \wedge \neg\varphi_5 \mapsto \{D_5\}$.
   $R_5 = \neg\varphi_4 \wedge \neg\varphi_5 \mapsto \{D_6\}$.
   $\varphi_4 = \mathsf{obs\text{-}event}(X_{\text{time}}, e_5(X_{\text{payment}})) \wedge t_{\text{deliver}} \leq X_{\text{time}}$.
   $e_5 = \lambda X_{\text{payment}}.(\text{“Pay Seller”}, \mathsf{transfer}(X_{\text{payment}}), \{\mathsf{buyer}\}, \{\mathsf{seller}\})$.
   $\varphi_5 = \mathsf{obs\text{-}event\text{-}before}(X_{\text{time}}, e_3)$.
   $D_5 : \mathsf{obs\text{-}total\text{-}paid}(X_{\text{time}}) = \mathsf{obs\text{-}total\text{-}paid}(X_{\text{time}} - 1) + X_{\text{payment}}$.
   $D_6 : \mathsf{obs\text{-}total\text{-}paid}(X_{\text{time}}) = \mathsf{obs\text{-}total\text{-}paid}(X_{\text{time}} - 1)$.

5. *Rules for Making Payments:*
   $R_6 = \neg\varphi_6 \mapsto \{A_3, R_7\}$.
   $\varphi_6 = \mathsf{obs\text{-}event\text{-}before}(t_{\text{deliver}} + 1, e_3)$.
   $A_3 = \mathbb{O}(e_5(200/2), [t_{\text{deliver}}, t_{\text{deliver}} + 1])$.
   $R_7 = \varphi_7 \wedge \neg\varphi_5 \mapsto \{D_7, R_8\}$.
   $\varphi_7 = \mathsf{obs\text{-}event}(X_{\text{time}}, e_5(200/2)) \wedge t_{\text{deliver}} \leq X_{\text{time}} \leq t_{\text{deliver}} + 1$.
   $D_7 : t_{\text{first}} = X_{\text{time}}$.
   $R_8 = \neg\varphi_8 \wedge \neg\varphi_5 \mapsto \{R_9\}$.
   $\varphi_8 = \mathsf{obs\text{-}event\text{-}during}([t_{\text{first}} + 1, t_{\text{deliver}} + 14], e_5(200/2))$.
   R_9 is defined below.

6. *Rules for Paying Fine for a Late Payment:*
   $R_9 = \neg\varphi_9 \mapsto \{A_4, A_5\}$.
   $\varphi_9 = \mathsf{obs\text{-}event\text{-}during}([t_{\text{deliver}} + 15, t_{\text{deliver}} + 30], e_5(200/2))$.
   $A_4 = \mathbb{O}(e_5(200/2), [t_{\text{deliver}} + 31, t_{\text{deliver}} + 44])$.
   $A_5 = \mathbb{O}(e_5(10\% * 200/2), [t_{\text{deliver}} + 31, t_{\text{deliver}} + 44])$.

t_{order}, t_{deliver}, t_{return}, and t_{first} are new constants of type Time. Each of the five events e_1, e_2, e_3, e_4, and e_5 are actions by one of the two parties. We assume that the events e_1, e_2, e_3, e_4 can happen at most once and the “Pay Seller” event e_5 can happen at most twice.

obs-total-paid(t) represents the total amount that the buyer has been observed to have paid the seller at time t. When rule R_4 is active, D_5 is generated. D_5 is used to add a payment to the total amount paid at the previous time point. D_5 and D_6 work together to record the happenings of the “Pay Seller” event e_5 in the timeline. obs-event-before(t, e), a constant of type Time $\times$ Event $\rightarrow$ Bool, represents the observation that the event e occurred on or before time t.

We identify that the buyer has the following options to choose from after he has accepted the printer and made the first payment:

1. Buyer makes a return within 14 days after the delivery is made.
2. Buyer makes the second payment within 14 days after the delivery made.
3. Buyer makes the second payment between 15 to 30 days after the delivery made.
4. Buyer makes the second payment with an additional fine between 31 to 44 days after the delivery made.

R_8 and R_9 work together as a reparation if the second payment is not made on time. Within 14 days the buyer has the first and second options to choose from. R_8 says if the first two options have not been chosen, then between 15 to 44 days the buyer is obligated to make the second payment. If it is paid late, which means R_9 is active, then an additional fine must be paid.

9 A Reasoning System

In this section we will sketch a reasoning system for FCL which is an extension of a proof system for simple type theory. Let $\bar{t}$ be an expression of type Time whose value is the time t, φ be a formula, Γ be a set of formulas, A be an agreement, R be a rule, and C be a contract. For a model $\mathcal{M}$ of FCL, $\mathcal{M} \vDash \varphi$ means $V^{\mathcal{M}}(\varphi) = \mathrm{T}$ and $\mathcal{M} \vDash \Gamma$ means $\mathcal{M} \vDash \psi$ for all $\psi \in \Gamma$.

A reasoning system for simple type theory (and other traditional logics) has a judgment of the form $\Gamma \vdash \varphi$ that asserts φ logically follows from Γ, i.e., $\mathcal{M} \vDash \Gamma$ implies $\mathcal{M} \vDash \varphi$ for all models $\mathcal{M}$ of FCL. Since constant definitions, agreements, and rules are not expressions of simple type theory, we need the following additional judgments in a reasoning system for FCL:

1. $\Gamma \vdash_{C,\bar{t}} \varphi$ asserts that φ logically follows from Γ and C at t, i.e., $\mathcal{M} \vDash \Gamma$ implies $\mathcal{M} \vDash \varphi$ for all models $\mathcal{M}$ of C at t.
2. $\mathsf{Agreement}[\Gamma, A, C, \bar{t}]$ asserts that A is in the state of C at t with respect to Γ, i.e., $\mathcal{M} \vDash \Gamma$ implies $A \in \mathsf{state}(C, \mathcal{M}, t)$ for all models $\mathcal{M}$ of FCL.
3. $\mathsf{Rule}[\Gamma, R, C, \bar{t}]$ asserts that R is in the state of the C at t with respect to Γ, i.e., $\mathcal{M} \vDash \Gamma$ implies $R \in \mathsf{state}(C, \mathcal{M}, t)$ for all models $\mathcal{M}$ of FCL.
4. $\mathsf{Satisfied}[\Gamma, A, C, \bar{t}]$ asserts that A is satisfied at t with respect to Γ, i.e., $\mathcal{M} \vDash \Gamma$ implies that A is satisfied in $\mathcal{M}$ at t for all models $\mathcal{M}$ of C at t.
5. $\mathsf{Violated}[\Gamma, A, C, \bar{t}]$ asserts that A is violated at t with respect to Γ, i.e., $\mathcal{M} \vDash \Gamma$ implies A is violated in $\mathcal{M}$ at t for all models $\mathcal{M}$ of C at t.
6. $\mathsf{Defunct}[\Gamma, R, C, \bar{t}]$ asserts that R is defunct at t with respect to Γ, i.e., $\mathcal{M} \vDash \Gamma$ implies R is defunct in $\mathcal{M}$ at t for all models $\mathcal{M}$ of C at t.
7. $\mathsf{Fulfilled}[\Gamma, C, \bar{t}]$ asserts that C is fulfilled at t with respect to Γ, i.e., $\mathcal{M} \vDash \Gamma$ implies C is fulfilled in $\mathcal{M}$ at t for all models $\mathcal{M}$ of C at t.
8. $\mathsf{Breached}[\Gamma, C, \bar{t}]$ asserts that C is breached at t with respect to Γ, i.e., $\mathcal{M} \vDash \Gamma$ implies C is breached in $\mathcal{M}$ at t for all models $\mathcal{M}$ of C at t.

The role of Γ is to specify the models in which C will be considered.

The reasoning system has several rules of inference including the usual rules of inference for simple type theory. There is a rule of inference that shows $\Gamma \vdash_{C,\bar{t}} \varphi$ extends $\Gamma \vdash \varphi$:

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash_{C,\bar{t}} \varphi}.$$

The rule of inference says that if φ follows from Γ, then φ follows from Γ and C at t for any contract C and time t.

The following three rules of inference show how a rule changes the state of a contract:

$$\frac{\mathsf{Rule}[\Gamma, R, C, \overline{t}], \Gamma \vdash_{C,\overline{t}} \varphi\sigma}{\Gamma \vdash_{C,\overline{t}+1} \psi_1\sigma, \ldots, \Gamma \vdash_{C,\overline{t}+1} \psi_k\sigma}$$

$$\frac{\mathsf{Rule}[\Gamma, R, C, \overline{t}], \Gamma \vdash_{C,\overline{t}} \varphi\sigma}{\mathsf{Agreement}[\Gamma, A_1\sigma, C, \overline{t}+1], \ldots, \mathsf{Agreement}[\Gamma, A_m\sigma, C, \overline{t}+1]}$$

$$\frac{\mathsf{Rule}[\Gamma, R, C, \overline{t}], \Gamma \vdash_{C,\overline{t}} \varphi\sigma}{\mathsf{Rule}[\Gamma, R_1\sigma, C, \overline{t}+1], \ldots, \mathsf{Rule}[\Gamma, R_n\sigma, C, \overline{t}+1]}$$

where

$$R = \varphi \mapsto \{\psi_1, \ldots, \psi_k, A_1, \ldots, A_m, R_1, \ldots, R_n\},$$

$\psi_1, \ldots, \psi_k$ are constant definitions, $A_1, \ldots, A_m$ are agreements, $R_1, \ldots, R_n$ are rules, and $\sigma \in \mathsf{sub}(\varphi, t)$

There is a rule of inference for satisfied agreements:

$$\frac{\mathsf{Agreement}[\Gamma, A, C, \overline{t}], \Gamma \vdash_{C,\overline{t}} A}{\mathsf{Satisfied}[\Gamma, A, C, \overline{t}]}.$$

There are similar rules of inference for violated agreements, defunct rules, fulfilled contracts, and breached contracts.

A reasoning system of this kind can be used to both prove statements about a contract and to simulate the unfolding of a contract over time. The latter is done by using Γ to specify the observations that are expected over the course of the contract. The reasoning system can be strengthened by introducing temporal operators that enable one to say, for example, that it follows from Γ that C will be *eventually* be fulfilled.

10 Related Work

Several formal languages for writing contracts have been proposed. Our language FCL is most closely related to the following work:

- S. L. Peyton Jones and J. M. Eber (J&E) [20,21].
- A. Goodchild, C. Herring, and Z. Milosevic (GHM) [12].
- G. Governatori and Z. Milosevic (G&M) [13,14,16].
- J. Andersen, E. Elsborg, F. Henglein, J.G. Simonsen, and C. Stefansen (AEHSS) [1].
- C. Prisacariu and G. Schneider (P&S) [10,23–26].
- P. Bahr, J. Berthold, M. Elsman (BBE) [5].
- LegalRuleML Technical Committee (TC) [2,3].

The domains of these approaches are varied: J&E's and BBE's works are restricted to financial contracts; GHM builds a domain-specific language for business contracts; AEHSS is concerned with formalizing commercial contracts; and the LegalRuleML TC focuses on the creation of machine-readable forms of the content of legal texts, such as legislation, regulations, contracts, and case law, for different concrete Web applications. Same as P&S's work, our proposed language FCL considers the formalization of general contracts that are agreements written by and for humans.

Several techniques are employed in the literature for developing a precise formal language for specifying contracts. Most of the techniques, such as those given in [12,13,16], belong to the event-condition-action (ECA) based scheme. GHM and G&M model contracts as sets of policies. A policy specifies that a legal entity is either forbidden or obliged to perform an action under certain event-based conditions. AEHSS provide an action-trace based language [1] to model contracts. J&E's functional programming based language [20,21] and BBE's cash-flow trace based approach [5] use the idea of observables to specify events. P&S introduce in [23–26] a contract language $\mathcal{CL}$ for expressing electronic contracts based on a combination of concepts from deontic, dynamic, and temporal logic. $\mathcal{CL}$ restricts deontic modalities to ought-to-do statements and adds the modalities of dynamic logic to be able to reason about what happens after an action is performed. Rather than providing a logical language for contracts, the LegalRuleML TC extends RuleML to provide a rule interchange language with formal features specific for the legal domain. This enables implementers to structure the contents of the legal texts in a machine-readable format by using the representation tools. Motivated by the ECA-based formalisms and the idea of observables, we introduce in FCL the concept of a *rule* that is (in its simplest form) a conditional agreement that depends on certain observations. The use of observables to determine both the meaning of a contract and how the meaning of the contract evolves over time provides a basis for monitoring the dynamic aspects of a contract.

Only P&S's $\mathcal{CL}$ language and LegalRuleML can specify reparation clauses. $\mathcal{CL}$ language incorporates the notions of contrary-to-duty and contrary-to-prohibition by explicitly attaching to the deontic modalities a reparation which is to be enforced in case of violations. LegalRuleML introduces in [2] a suborder list that is a list of deontic formulas to model penalties. We think $\mathcal{CL}$'s and LegalRuleML's use of only contrary-to-duty obligations to recover a contract when it is breached is too limited. There is no provision provided for recovery from technical or business-related issues. In FCL, we interpret an agreement in a contract in terms of the deontic concepts of *obligation* and *prohibition*. These concepts are applied in expressions to actions that are executed by the parties of the contract. Thus, the concepts express what a party *ought to do* and or *ought not do*. FCL rules can also be used for reparational purposes when an agreement is violated (see Subsect. 6.2).

With the exception of approaches provided by AEHSS, BBE, P&S, and LegalRuleML TC, all of the languages above are informal. The work of both AEHSS and P&S include a trace-based reduction semantics model for contracts. These two approaches provide a run-time monitoring of the fulfillment and breach of a contract since the state of a contract at a time is determined by the events that have happened. LegalRuleML utilizes the defeasible deontic logic to reason about violations of obligations. Both GHM's work and G&M's work lack a formal semantics and a reasoning system even though they provide a good framework for monitoring contracts. The semantics provided by J&E in [21] is based on stochastic processes. J&E's approach provides the ability to perform compositional analysis of monetary values of contracts. This work can estimate the expected value of financial contracts. BBE's trace-based semantics allows the modification of a contract according to the passage of time and the values of observables. But since both of the approaches provided by J&E and BBE pay more attention to finding the monetary value of contracts, they consider the semantic meaning of a contract to be its cash-flow gain or loss, which is too limited for general contracts from our point of view. We find this lack of work on formal semantics surprising since one of the main benefits of defining a contract language to be formal is to enable the language to have a precise, unambiguous semantics.

Although the languages of AEHSS and P&S provide a formal mathematical model for contracts with a formal semantics and are able to express some important features of contracts, they are not as expressive as FCL. For example, in the case where a contract is breached, the monitor should not only report a breach of contract, but also who among the contract parties is responsible (blame assignment). Except for the languages provided by BBE and LegalRuleML TC, all the other contracts covered by these approaches, including the work of AEHSS and P&S, are two-party contracts in which the parties are implicit. These approaches are not able to determine who is to be blamed when a contract is breached. Our proposed language provides explicit participants and thus provides the possibility of having contracts with both an unrestricted number of parties and with blame assignment.

In addition, because time constraints are implicit in P&S's $\mathcal{CL}$ language, it only has relative deadlines where one party's commitment to do something depends on when the other party has performed an action. Our proposed language FCL has not only relative temporal constraints, but also absolute temporal constraints.

11 Conclusion and Future Work

In this paper we have presented FCL, a formal language for writing contracts that may contain temporally based conditions. Changes to the meaning of a FCL contract are triggered when the conditions in it become true. FCL admits agreements that correspond to the deontic notions of obligation and prohibition, can express conditions that depend on events and other observables, and include condition-based rules to define new constants and introduce new agreements and rules. We have sketched a reasoning system for FCL. To our knowledge, no other formal contract language is as expressive as FCL.

FCL offers three advantages to the contract writer. First, since FCL has a precise semantics, contracts written in FCL have an unambiguous meaning. Since the underlying logic of FCL is simple type theory, the semantics of contracts written in FCL is based on very well understood ideas and the logical tools for writing contracts in FCL are very expressive. Third, since FCL is a formal language, software-implemented formal methods can be used to assist in the writing and analysis of FCL contracts. In particular, we can use software tools to check whether an action in a contract has been performed or not, to report whether a contract has been fulfilled or violated, to compute the value of a contract, etc. We can also use software tools to reason about possible future outcomes of a contract and about the relationship between different contracts.

Our future work will include (1) extending the design of FCL, (2) writing several additional examples of contracts in FCL, (3) finishing the development of a reasoning system for FCL, (4) designing a module system for building contracts out of contract modules, and (5) integrating FCL with contract law and regulations. We will also validate FCL by implementing it in Agda [7,18,19], a dependently typed functional programming language. In a future paper, we will give a full presentation of FCL and its implementation in Agda.

Acknowledgments. The authors are grateful to the reviewers for their comments and suggestions. This research was supported by NSERC.

References

1. Andersen, J., Elsborg, E., Henglein, F., Simonsen, J., Stefansen, C.: Compositional specification of commercial contracts. Int. J. Softw. Tools Technol. Transfer (STTT) **8**(6), 485–516 (2006)
2. Athan, T., Boley, H., Governatori, G., Palmirani, M., Paschke, A., Wyner, A.: OASIS LegalRuleML. In: Francesconi, E., Verheij, B. (eds.) ICAIL, pp. 3–12. ACM (2013)
3. Athan, T., Governatori, G., Palmirani, M., Paschke, A., Wyner, A.Z.: LegalRuleML: design principles and foundations. In: Faber, W., Pashke, A. (eds.) The 11th Reasoning Web Summer School, pp. 151–188. Springer, Germany, July 2015
4. Attorney, R.S.: Contracts: The Essential Business Desk Reference. Nolo (2010)
5. Bahr, P., Berthold, J., Elsman, M.: Certified symbolic management of financial multi-party contracts. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, 1-3 September 2015, pp. 315–327 (2015)
6. Blum, B.A.: Contracts: Examples and Explanations, 4th edn. Aspen Publishers, New York (2007)
7. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda — a functional language with dependent types. In: TPHOLs, vol. 9, pp. 73–78. Springer (2009)
8. Farmer, W.M.: The seven virtues of simple type theory. J. Appl. Logic **6**, 267–286 (2008)
9. Farmer, W.M., Hu, Q.: A formal language for writing contracts. In: 2016 IEEE 17th International Conference on Information Reuse and Integration (IRI 2016), pp. 134–141. IEEE (2016)
10. Fenech, S., Pace, G.J., Schneider, G.: Automatic conflict detection on contracts. In: International Colloquium on Theoretical Aspects of Computing, pp. 200–214. Springer (2009)

11. Finan, M.B.: A discussion of financial economics in actuarial models: a preparation for exam MFE/3F (2015). Prepared for Arkansas Tech University
12. Goodchild, A., Herring, C., Milosevic, Z.: Business contracts for B2B. In: Proceedings of the CAISE00 Workshop on Infrastructure for Dynamic Business-to-Business Service Outsourcing, Stockholm, Sweden (2000)
13. Governatori, G., Milosevic, Z.: A formal analysis of a business contract language. Int. J. Coop. Inf. Syst. **15**(04), 659–685 (2006)
14. Governatori, G., Rotolo, A.: Logic of violations: a gentzen system for reasoning with contrary-to-duty obligations. Australas. J. Logic **4**, 193–215 (2005)
15. Hvitved, T., Klaedtke, F., Zălinescu, E.: A trace-based model for multiparty contracts. J. Logic Algebraic Program. 81, 72–98 (2012)
16. Linington, P.F., Milosevic, Z., Cole, J., Gibson, S., Kulkarni, S., Neal, S.: A unified behavioural model and a contract language for extended enterprise. Data Knowl. Eng. **51**(1), 5–29 (2004)
17. McNamara, P.: Deontic logic. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Winter 2014 (2014)
18. Norell, U.: Towards a Practical Programming Language based on Dependent Type Theory. Ph.D. thesis, Chalmers University of Technology (2007)
19. Norell, U.: Dependently typed programming in Agda. In: Kennedy, A., Ahmed, A. (eds.) Proceedings of the 4th International Workshop on Types in Language Design and Implementation, TLDI 2009, vol. 2, pp. 1–2. ACM, New York (2009)
20. Peyton Jones, S.L.: Composing contracts: an adventure in financial engineering. In: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity. Lecture Notes in Computer Science, vol. 2021, p. 435. Springer (2001)
21. Peyton Jones, S.L., Eber, J.M.: How to write a financial contract. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming. Cornerstones in Computing, pp. 105–130. Palgrave (2003)
22. Poole, J.: Textbook on Contract Law, 11th edn. Oxford University Press, Oxford (2012)
23. Prisacariu, C., Schneider, G.: An algebraic structure for the action-based contract language cl-theoretical results. Technical report 361, Department of Informatics, University of Oslo, Oslo, Norway (2007)
24. Prisacariu, C., Schneider, G.: A formal language for electronic contracts. In: International Conference on Formal Methods for Open Object-Based Distributed Systems, pp. 174–189. Springer (2007)
25. Prisacariu, C., Schneider, G.: Towards a formal definition of electronic contracts. Technical Report 348, Department of Informatics, University of Oslo, Oslo, Norway (2007)
26. Prisacariu, C., Schneider, G.: A dynamic deontic logic for complex contracts. J. Logic Algebraic Program. **81**, 458–490 (2012)

Operational Semantics for the Rigorous Analysis of Distributed Systems

Mohammed S. Al-Mahfoudh(✉), Ganesh Gopalakrishnan, and Ryan Stutsman

School of Computing, University of Utah, Salt Lake City, UT 84112, USA
{mahfoudh,ganesh,stutsman}@cs.utah.edu

Abstract. The development of distributed systems based on poorly specified abstractions can hinder unambiguous understanding and the creation of common formal analysis methods. In this paper, we outline the design of a system modeling language called DS2, and point out how its primitives are well matched with concerns that naturally arise during distributed system design. We present an operational semantics for DS2 as well as results from an ongoing Scala-based implementation slated to support a variety of state-space exploration techniques. The driving goals of this project are to: (1) provide a prototyping framework within which complex distributed system protocols can be stated and modeled without breaking the primitives down to low level ones, and (2) drive the development of interesting and distributed system-relevant property checking methods (e.g., linearizability).

Keywords: Distributed systems · Operational semantics · Scheduling · Concurrency · Actors

1 Introduction

Distributed systems, both large scale and small, are now critically important in nearly every aspect of computing and in our lives. From embedded micro-controllers to data centers and web applications that span multiple geographic locations, more developers are building distributed systems than ever before. The ability to rapidly construct and deploy correctly functioning distributed systems is a growing necessity; unfortunately, this is hard even for experts. Non-determinism, weak update consistency, and complex failure handling protocols push far beyond what manual reasoning can grapple with, making it hard to offer *basic* safety guarantees.

The formal methods research community has begun responding to some of these challenges associated with building distributed systems. Recent efforts have formalized several famously subtle algorithms [25] and generated correct, synthesized implementations [13,29]. While these efforts represent significant steps forward, their wider applicability is limited. For example, approaches that rely on automated theorem proving [29] are beyond the expertise of all but a handful of academics and tend to abstract away from situation-specific complexities.

S.H. Rubin and T. Bouabana-Tebibel (eds.), *Quality Software Through Reuse and Integration*, Advances in Intelligent Systems and Computing 561,
DOI 10.1007/978-3-319-56157-8_10

Approaches that require developers to model systems using predicate logic [22] or model-checking frameworks [12,16] provide only limited reasoning capabilities and do not provide the requisite higher-level abstractions necessary for intuitive modeling.

Concurrently with all these developments, new languages and frameworks are continually emerging to help developers rapidly create new distributed systems. Languages such as Go and Rust and frameworks such as Akka are enabling developers to build and deploy distributed systems more easily. However, these approaches provide no formal basis for reasoning about the safety and correctness of the resulting systems. The guarantees that they do provide are informally documented, are non-portable, and low-level. Consequently, the developers' ability to build and deploy complex systems has vastly outpaced their ability to *reason* about their behaviors and detect design flaws. There is however an opportunity here: by combining a simple and expressive programming model with strong formal guarantees, one can achieve the best of both worlds. With that goal in mind, we are developing a domain-specific language for distributed systems called DS2[1].

DS2 is being developed to allow developers specify, test, verify, and synthesize correct distributed systems built on a clear operational semantics. DS2 uses an Actor-based concurrency model [3,11,14] that is compatible with the popular Akka distributed systems programming framework [1]. A working prototype of DS2 system written in Scala exists, and many driving examples are being ported using a front-end. Preliminary results from our effort appear in DS2's official website [9]. This paper details the aforementioned preliminary work's operational semantics, both its normal operation and its faults model.

While developing some of the central features underlying state-space exploration of distributed systems[2] expressed in DS2, we realized first-hand how treacherous the corner-cases can be. We detail some of these subtleties in Sect. 7. This gave further impetus to our work on writing down a clear operational semantics.

The need for clear guiding semantics underlying distributed systems is further exemplified by Chord's [26] erroneous operation [31] that was discovered with help of a manually written model of it in an external model checker. In short, having a semantic basis helps carry out automated semantics-guided verification, supports more objective comparisons between various efforts, and helps prepare the community to handle newer protocols being designed. A variety of formal (e.g., linearizability checking [6]) and semi-formal (e.g., lineage-driven fault-injection [24]) techniques can also build on a clear semantics.

Background: It is important to point out some central characteristics of distributed systems, over and above those present in shared-memory concurrency (e.g., P-Threads) or traditional message-passing based parallel programming (e.g., using MPI). Distributed systems ingest all these concurrency-related subtleties, and additionally present other challenges:

[1] Domain-Specific/Declarative-Specification of Distributed Systems.

[2] Our focus is towards networked, asynchronous, deterministic, and non-Byzantine distributed systems. However, the model is *easily* extensible to include Byzantine distributed systems, too.

– Weak (eventual [5]) consistency is the norm rather than the exception. This adds to the non-intuitiveness of distributed systems behaviors.
– Given that faulty operation is the norm rather than the exception, processes of a distributed system hover on different planes of operation at different times (i.e. fault-free or faulty planes). Each time a fault occurs, the correctness guarantees are subsetted to a core set of primitive ones. When faults are correctly handled and normalcy returns, guarantees are elevated. A designer must be empowered with the ability to simulate these fault/recovery driven transitions, and check for the right level of guarantees being delivered.

2 Contributions

A key feature of our approach is the adoption of the simple *strategy* OO design pattern [27], wherein the scheduler is the algorithm and the distributed system is the context. This design approach provides both flexibility and extensibility both with respect to actual algorithms, as well as implementations. Another feature is that our design of DS2 facilitates the specification of normal behaviors, and introduces – in a layered manner – various scheduling options that mimic faulty behaviors. This matches the fault/recovery-driven transitions referred to earlier. We now present specific primitives of DS2 that facilitate the creation, extension, and systematic exploration of candidate distributed system designs:

– *Locking mechanism:* It allows an agent to control when to (and when not to) receive messages, to model a single process disconnect.
– *Message dropping:* It helps model failures such as dropped packets and/or network partitioning.
– *Futures:* At the implementation level, DS2 employs a mediator agent (temporary agent) to model an *ask* pattern. The use of a mediator agent removes the need to broadcast the handle to the replier, thus freeing the asker from manually handling the future resolution. This is inspired by the actual Akka [1] implementation.
– *Mixins:* Mixins are known as traits in Scala. They enable the designer to incorporate variations to fault models into a scheduler elegantly. This approach provides flexibility for algorithm designers, while at the same time shielding users from distributed systems complexities. This approach also helps avoid making *a priori* assumptions about fault models.
– *State capture:* capturing the global state and resumption from such captured global states allows us to develop backtracking algorithms, on-the-fly scheduler switching, and parallel schedules exploration.
– *Expressive power-wise:* DS2 [4] has a model part (that facilitates state space exploration) and a language part (a DSL for concisely specifying distributed systems). In this work, we focus on the model that facilitates the state space exploration and distributed systems analyses. It provides a rich set of primitives to support flexible state-space exploration methods (as compared with more standard model-checking notations such as TLA+ and Promela).

Our approach is more in line with the needs of a real distributed systems designer. In contrast, some prior language designs (notably Actors [3]), have emphasized novel state-space exploration algorithms [28], and have not emphasized (to the same extent) the creation of a guiding operational semantics or emphasis on key primitives as we summarized.

3 Overview

In this section, we give a brief overview of how our design choices fit together synergistically. In Sect. 4, we present a more detailed example, walking through most of the operational semantics rules to address the details.

In a real world example, when a distributed system has been constructed, normally one or more agents comprising it should be bootstrapped, i.e. started. It is exactly the same for our model, however with the scheduler taking control of how the state of the system should evolve, by deciding what events happen when and where.

After bootstrapping an agent, we arrange for it to have a `Start` message in its input queue. The scheduler dequeues this message, finds the matching action from the agent's reactions map, and makes a copy of that action-template and instantiates its relevant parameters. After that, the scheduler *schedules* the action into its task queue (making it a task). The scheduler then can choose whether to schedule something from another agent or merely resume executing the scheduled task(s). Consuming a statement leads to that statement being enqueued into the scheduler's consume queue. We employ a consume queue to expose interleaving effects of statements coming from different tasks scheduled by different agents. The scheduler then executes each statement, removing it from the front of the consume queue. This cycle repeats until some specified stopping criteria are met.

The aforesaid simple design gives the scheduler the ability to model different planes/levels of faulty behaviors in isolation (or combination, if chosen). Examples of situations that can be modeled include reordering, duplication, and dropping of messages; these are modeled by manipulating the agent's queue. Disconnects of single processes can be modeled using *locking*, and network partitioning can be modeled by *message dropping* from selected agents queue's, hence simulating missing updates. The scheduler consume queue can be manipulated to simulate different interleavings as well as delays of execution, forcing different bug scenarios. Similarly, a crash in a node (agent), and injecting a fault or a message that causes a fault can be simulated.

The rest of the paper is organized in the following manner: an illustrative example is given in Sect. 4, followed by the walkthrough Sect. 5 of the operational semantics rules that are stated in Fig. 2, then the faults operational semantics are explained in Sect. 6. A real example of a bug discovery in our project is discussed briefly in Sect. 7. After that, we present related work in Sect. 8 followed by the concluding remarks in Sect. 9.

4 Walk Through Example

A distributed system is constructed by the code in Listing 1.1. We begin by creating a distributed system `ds`, a client `c` and a server `s`. Then a reaction is added to the client in lines 6 through 10. Two reactions are then added to the server in lines 12 through 16. Finally, the client and server are added to the distributed system that, in turn, is attached to the basic scheduler (*basic* in the sense of being controlled by the user).

```
1  val ds = new DistributedSystem("Echo ack")
   val s = new Agent("Server")
3  val c = new Agent("Client")
   val act1, act2, act3 = new Action
5  // Client setup
   act1 + Statement(UNLOCK,c) // unlocks the agent incoming q
7  act1 + Statement(ASK,c,new Message("Show","Hello!"),s, "vn")
   act1 + Statement(GET,c,"vn","vn2")
9  act1 + Statement(println("I'm Happy!"))
   c.R("Start") = act1 // (Start, act1) to reactions map
11 // Server setup
   act2 + Statement(UNLOCK, s)
13 act2 + Statement(println("Greetings!"))
   act3 + Statement((m:Message,a:Agent)=>println(m.p))
15 act3 + Statement((m:Message,a:Agent)=>send(s,m(p = true),m.s))
   s.R("Start") = act2 ; s.R("Show") = act3
17 ds += Set(s,c) // adding agents to system
   ds.attach(BasicScheduler)
```

Listing 1.1: An example distributed system, echo server-client interaction with additional blocking for an acknowledgement on client

One execution of such a distributed system is shown in Listing 1.2. We deliberately chose a schedule that leads to a large number of rule-firings. The listing is explained via the comments, and Fig. 1 visualizes the resulting sequence of states step-by-step.

A design-time question might be: what schedule might lead to a client avoiding blocking on a future? The answer is to swap the consuming of the resolving send (on line 17 of Listing 1.2), with consuming the blocking get (on line 16 of the same listing). Another intent may be to exhibit a deadlock by the client. We then keep the same schedule, but we remove all statements and actions, consumed and scheduled respectively, from the server, and reset the server state (i.e. locking the server, then making q $= \epsilon$, and making its local state $\mathscr{L}$ empty) before it executes the resolving send. By doing the latter, we simulated a crash of the server and there is no way that client gets its future resolved, ending it in the simplest forms of a deadlock. One last attempt is to simulate a message drop for a message sent by the same resolving send to resolve the client's future (say we drop RF from client's queue, since getting the messages into the queue does not mean it is delivered, but rather means it is *in flight* and only considered delivered after it gets scheduled and/or handled by the receiving agent).

```
val sch = ds.scheduler
sch.boot(s); sch.boot(c) // sends Start msg to s and to c
sch.schedule(s) // schedule start task from s
sch.schedule(c) // schedule start task from c
sch.consume(s)  // consume UNLOCK stmt from s task
sch.consume(s)  // consume "greeting" stmt from s task
sch.consume(c)  // consume UNLOCK stmt from c task
sch.consume(c)  // consume ASK stmt from c task
sch.executeOne  // UNLOCK s stmt, IsLocked(s) == false
sch.executeOne  // "greeting" s stmt
sch.executeOne  // UNLOCK c stmt, IsLocked(c) == false
sch.executeOne  // ASK s stmt, T = {t} temporary agent
                // and s.q == [Show("Hello",s=t)]
sch.schedule(s) // schedule "Show" task from s
sch.consume(s)  // consume print("Hello") stmt
sch.consume(c)  // consume GET stmt from c task
sch.consume(s)  // consume resolving send(..) stmt
                // note GET blocks, then it is resolved
sch.consume(c)  // consume "happy" stmt from c task
sch.executeOne  // s print("Hello")
sch.executeOne  // c blocks on GET, does not progress
                // putting back all stmts after it
                // from cq back to front of task.xq in order
sch.executeOne  // resolving send(..), t.q != empty
                // things happen to t.L("vn") future resolved
                // and then c.q = [RF(f,s=s)], note sender
                // is s, not t
sch.handle(c)   // handling the RF message, unblocking c
sch.consume(c)  // consuming GET from c again
sch.consume(c)  // consuming "happy" stmt from c
sch.executeOne  // R GET c stmt, won't block (resolved)
                // c.L("vn2") = c.L("vn").val
sch.executeOne  // print("I'm happy")
// DONE happy schedule, other schedules are not this happy
```

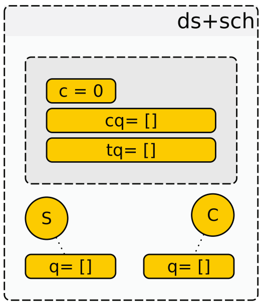
Listing 1.2: Example schedule invoking SCHEDULE, CONSUME, ASK, BLK-GET, R-SEND, and R-GET rules. The equivalent visualization of this execution is shown in Figure 1

In addition, fault assumptions can be enabled/disabled selectively to affect how a scheduler explores a distributed system, i.e. the operation of a scheduler is *not* disconnected from these assumptions. In our implementation, we have these assumptions implemented in the form of mixin Scala traits. When these traits are mixed into a scheduler they change what different faults are/not allowed to be simulated by the scheduler. Even better, a developer can replace, extend, re-use and/or override these mixins by providing their own scheduler-specific implementations that, in turn, enable their schedulers inject/simulate faults in a very specific manner with respect to their algorithms.

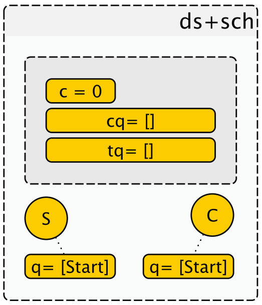
All these are illustrations that highlight the flexibility offered by the DS2 approach in creating faulty situations, and forcing a plethora of semi-formal state exploration methods to cover them.

5 Operational Semantics

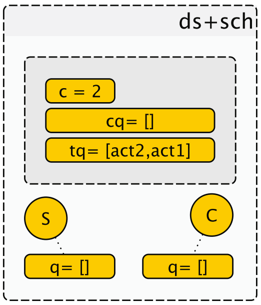
Despite DS2's simplicity, creating a concise semantics for it proved to be a challenge. Our initial operational semantics spanned dozens of pages [9], which led us to invent several abstract state predicates. The result is a concise semantics expressed in eight simple rules, and complemented with another eight fault-rules Sect. 6, that still retain the full expressiveness that real-world developers



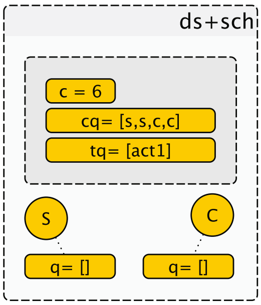
(a) Initial State



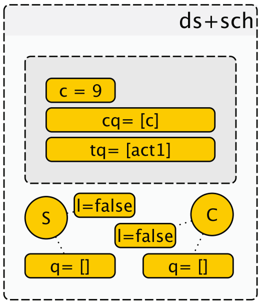
(b) Executed line 2



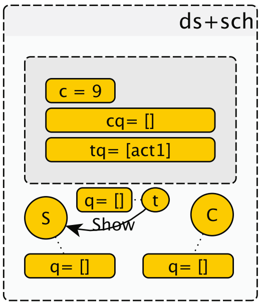
(c) Executed lines 3,4



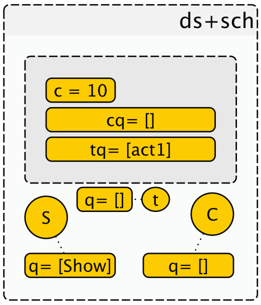
(d) Executed lines 5-8



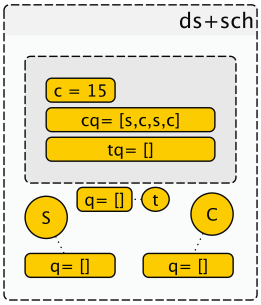
(e) Executed lines 9-11



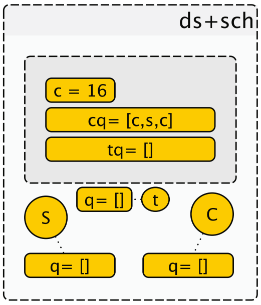
(f) Executing line 12



(g) Executed line 12

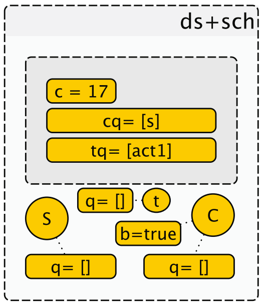


(h) Executed 14-19

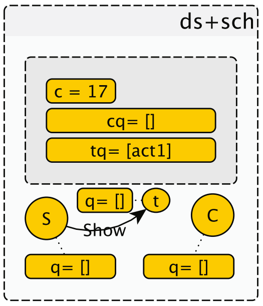


(i) Executed 20

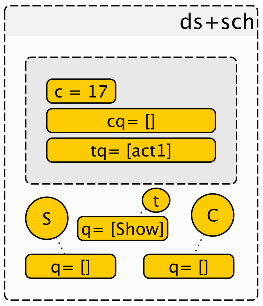
Fig. 1. Example run invoking majority of operational semantics rules. Subfigures refer to line numbers in Listing 1.2



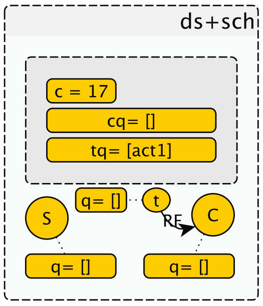
(j) Executed 21



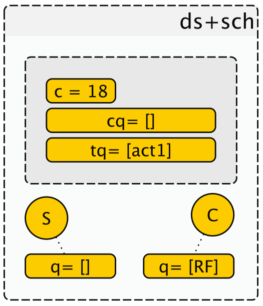
(k) Executing 24-(1/4)



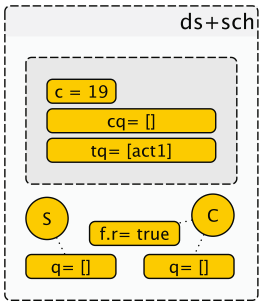
(l) Executing 24-(2/4)



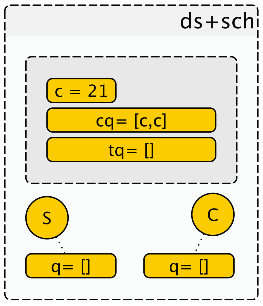
(m) Executing 24-(3/4)



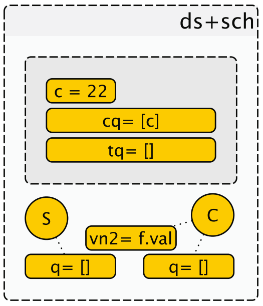
(n) Executing 24-(4/4)



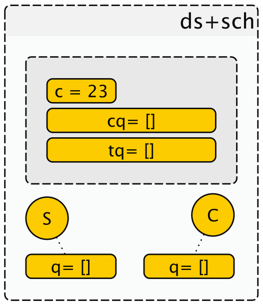
(o) Executed line 28



(p) Executed 29,30



(q) Executed 31



(r) Executed 33

Fig. 1. (*continued*)

demand. For example, it provides both asynchronous communication and synchronization primitives. At the same time, it forms the minimal rule set needed to understand and reason about communication and synchronization patterns (Sect. 5.2). This conciseness gives us confidence that our operational semantics and model will benefit designers who seek to build formal method tools and those seeking to model and understand new and existing distributed protocols.

$$\text{SCHEDULE}\;\frac{\begin{array}{c}\mathit{ChoseSchedule}(\Sigma,\alpha)\wedge\gamma_\alpha\ \mathit{as}\ \langle\mu,\alpha,\mathit{stmts},\zeta_x\rangle=\mathscr{R}(\mu)\wedge\\ \mathit{stmts}_\gamma\neq\epsilon\wedge\zeta_{x,\gamma}=\mathit{stmts}_\gamma\end{array}}{\langle\mathscr{A}(\alpha(\mu.q,,\neg b,\mathscr{R},,,)),\mathscr{T},\Sigma(,\zeta_t,)\rangle\rightarrow\langle\mathscr{A}(\alpha(q,,\neg b,\mathscr{R},,,)),\mathscr{T},\Sigma(,\zeta_t.\gamma_\alpha,)\rangle}$$

$$\text{CONSUME}\;\frac{\mathit{ChoseConsume}(\Sigma,\alpha)}{\begin{array}{c}\langle\mathscr{A}(\alpha(,,\neg b,,,,)),\mathscr{T},\Sigma(,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,,,s.\zeta_x),\ldots),\zeta_c)\rangle\rightarrow\\ \langle\mathscr{A}(\alpha(,,\neg b,,,,)),\mathscr{T},\Sigma(,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,,,\zeta_x),\ldots),\zeta_c.s)\rangle\end{array}}$$

$$\text{SEND}\;\frac{\begin{array}{c}\mathit{IsSnd}(s)\wedge\langle\alpha_s,\mu,\alpha_d\rangle=\mathit{Involved}(s)\wedge\neg\mathit{IsRSnd}(\mathscr{T},s)\wedge(p=p')\wedge\\ \mathit{ChoseExOne}(\Sigma)\end{array}}{\begin{array}{c}\langle\mathscr{A}(\{\alpha_s(,,\neg b,,,,p),\alpha_d(q,\neg l,,,,,p')\}),\mathscr{T},\Sigma(c,,s.\zeta_c)\rangle\rightarrow\\ \langle\mathscr{A}(\{\alpha_s(,,\neg b,,,,p),\alpha_d(q.\mu,\neg l,,,,,p')\}),\mathscr{T},\Sigma(c+1,,\zeta_c)\rangle\end{array}}$$

$$\text{ASK}\;\frac{\begin{array}{c}\mathit{IsAsk}(s)\wedge\langle\alpha_s,\mu,\alpha_d,\mathit{vn}\rangle=\mathit{Involved}(s)\wedge(p=p')\wedge\mathit{ChoseExOne}(\Sigma)\wedge\\ f=\mathit{fresh}(\mathit{Future})\wedge\alpha_t=\mathit{fresh}(\mathit{Agent})\wedge\mathit{where}=\mathit{fresh}(\mathit{Ids})\end{array}}{\begin{array}{c}\langle\mathscr{A}(\{\alpha_s(,,\neg b,,,,p),\alpha_d(q,\neg l,,,,,p')\}),\mathscr{T},\Sigma(c,,s.\zeta_c)\rangle\rightarrow\\ \langle\mathscr{A}(\alpha_s(,,,,,\mathscr{L}\cup(\mathit{vn},f),p),\alpha_d(q.\mu(\alpha_t,),\neg l,,,,,p')),\mathscr{T}\cup\\ \alpha_t(,\neg l,\neg b,,,\mathscr{L}\cup\{(\mathit{vn},f),(\mathit{where},\alpha_s)\},p'),\Sigma(c+1,,\zeta_c)\rangle\end{array}}$$

$$\text{R-SEND}\;\frac{\begin{array}{c}\mathit{IsSnd}(s)\wedge\langle\alpha_s,\mu,\alpha_t\rangle=\mathit{Involved}(s)\wedge\mathit{IsRSnd}(\mathscr{T},s)\wedge\\ (p=p')\wedge\mathit{ChoseExOne}(\Sigma)\wedge\mathit{RF}=\mathit{fresh}(\mathit{Message})\end{array}}{\begin{array}{c}\langle\mathscr{A}(\alpha_s(,,\neg b,,,,p),\alpha_d(\mathit{RF}^*.q,\neg l,,,,\mathscr{L}\cup(\mathit{vn},f),p')),\mathscr{T}\cup\\ \alpha_t(,\neg l,,,,\mathscr{L}\cup\{(\mathit{vn},f),(\mathit{where},\alpha_d)\},p'),\Sigma(c,,s.\zeta_c)\rangle\rightarrow\\ \langle\mathscr{A}(\alpha_s(,,\neg b,,,,p),\alpha_d(\mathit{RF}^*.\mathit{RF}(\alpha_s,f(\mathit{true},\mu[p_0])).q,),\neg l,,,,\mathscr{L}\cup(\mathit{vn},f),p'),\\ \mathscr{T}\setminus\{\alpha_t(,,,,,\mathscr{L}\cup\{(\mathit{vn},f(\mathit{true},\mu[p_0])),(\mathit{where},\alpha_d)\},p')\},\Sigma(c+1,,\zeta_c)\rangle\end{array}}$$

$$\text{R-GET}\;\frac{\begin{array}{c}(\mathit{IsGet}(s)\vee\mathit{IsTGet}(s))\wedge\\ (\langle\alpha,\mathit{vn},\mathit{vn}2\rangle=\mathit{Involved}(s)\vee\langle\alpha,\mathit{vn},\mathit{vn}2,\mathit{to}\rangle=\mathit{Involved}(s))\wedge\mathit{ChoseExOne}(\Sigma)\end{array}}{\begin{array}{c}\langle\mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup(\mathit{vn},f(\mathit{true},\mathit{val})),)),\mathscr{T},\Sigma(c,,s.\zeta_c)\rangle\rightarrow\\ \langle\mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup\{(\mathit{vn},f(\mathit{true},\mathit{val})),(\mathit{vn}2,\mathit{val}_f)\},)),\mathscr{T},\Sigma(c+1,,\zeta_c)\rangle\end{array}}$$

$$\text{BLK-GET}\;\frac{\begin{array}{c}\mathit{IsGet}(s)\wedge\langle\alpha,\mathit{vn},\mathit{vn}2\rangle=\mathit{Involved}(s)\wedge\\ \mathit{ChoseExOne}(\Sigma)\wedge\mathit{ss}=\mathit{PreEmpted}(\Sigma,\alpha)\end{array}}{\begin{array}{c}\langle\mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup(\mathit{vn},f(\mathit{false},)),)),\mathscr{T},\Sigma(c,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,\alpha,,\zeta_x),\ldots),s.\zeta_c)\rangle\rightarrow\\ \langle\mathscr{A}(\alpha(,,b,,,\mathscr{L}\cup(\mathit{vn},f(\mathit{false},)),)),\mathscr{T},\Sigma(c+1,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,\alpha,,s.\mathit{ss}.\zeta_x),\ldots),\zeta_c\setminus\mathit{ss})\rangle\end{array}}$$

$$\text{T-GET}\;\frac{\begin{array}{c}\mathit{IsTGet}(s)\wedge\langle\alpha,\mathit{vn},\mathit{vn}2,\mathit{to}\rangle=\mathit{Involved}(s)\wedge\\ \mathit{ChoseExOne}(\Sigma)\wedge\mathit{ss}=\mathit{PreEmpted}(\Sigma,\alpha)\wedge\mathit{to}\leq 0\end{array}}{\begin{array}{c}\langle\mathscr{A}(\alpha(,,,,,\mathscr{L}\cup(\mathit{vn},f(\mathit{false},)),)),\mathscr{T},\Sigma(c,,s.\zeta_c)\rangle\rightarrow\\ \langle\mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup(\mathit{vn},f(\mathit{false},)),)),\mathscr{T},\Sigma(c+1,,\zeta_c.\mathit{ss})\rangle\end{array}}$$

Fig. 2. DS2 operational semantics

We achieved this parsimony by defining a set of structures representing the state of a distributed system, a set of predicates to de-clutter the rules from mathematical details, and a set of conventions to make our presentation intuitive.

5.1 Structures

Now, we introduce the core structures of our semantics. We use the convention $(,,,p,q,,r,\ldots)$ to indicate three don't care arguments followed by p and q, then one don't care followed by r, and the tail sequence is a don't care.

Distributed System is a tuple $\langle \mathscr{A}, \mathscr{T}, \Sigma \rangle$ with the set of agents in a distributed system $\mathscr{A}$, the set of all *temporary* agents $\mathscr{T}$ (initially empty; each such agent handles resolving a single future, and then disappears), and a scheduler Σ. We will first elaborate the primitives in our model, and then return to the components of the distributed system.

Message is a tuple $\langle s, p_0, \ldots, p_n \rangle$ where s is the sender agent, and $p_0, \ldots, p_n$ is the payload of the message. The set of messages is symbolized by $\mathscr{M}$.

Statement is a wrapper around the code to execute. It is a tuple $\langle \gamma, code, k, p^* \rangle$ where γ is the action containing this statement, *code* is the actual code to execute, k indicates the kind of a statement e.g. **Send** for send statement, and p^* are the parameters of the statement. A statement holds a reference to its containing action γ to provide access to the message m that invoked the action and agent a whose local state could be modified/accessed by the statement. In addition, in case a statement was preempted, it is known where it should be put back (to the front of the to-execute queue of the action γ); a walk-through of the rules in Sect. 5.4 will clarify this.

Action is essentially the sequence of statements to execute. More specifically, it is a tuple $\langle m, a, stmts, \zeta_x \rangle$ where a is the agent containing this action (whose local state may get modified/accessed by this action's statements), m is the message that invoked this action, *stmts* is a sequence of statements or the template. ζ_x is a reference to the *to-execute queue* of statements. The set of all actions is Γ.

Timed Action is an action associated with two values of time. It is the tuple $\langle t1, t2, \gamma \rangle$ where γ is scheduled to execute every $t1$ time with tolerance of $t2$ amount of time, usually used to model *heart beat*. The tolerance time $t2$ is there to add delay-tolerance for message delivery, since it cannot be predicted.

Future is a synchronization construct, which is a tuple $\langle r, val \rangle$ where r is a boolean indicating whether the future is resolved (defaults to $false$) and val is the value resolving this future (defaults to $\perp$, or no value). A future object is returned by the **Ask** statement type, as will be presented shortly.

Agent is a communicating autonomous process. It is a tuple $\langle q, l, b, \mathscr{R}, \tau, \mathscr{L}, p \rangle$ where q is its receive queue, the *reactions* of the agent $\mathscr{R} : \mathscr{M} \rightarrow \Gamma$, timed actions set $\tau = \{\langle t1, t2, \gamma \rangle\}$, $\mathscr{L}$ is the local state of an agent $\mathscr{L} : Ids \rightarrow Vals$. Blocked b and locked l are flags whose initial values are `false` and `true`, respectively. A blocked agent cannot execute statements, consume, or schedule tasks, an agent is blocked if a future it tries to access is not resolved. Locked l is used by the scheduler to tell if the process is receiving messages or not, e.g. its server socket is not open, to model a disconnect. In order to model *network partitioning*, however, the p is the partition id to which this agent belongs, initially the same for ALL agents. Many or all agents can share the same value, if they are in the same network partition. Otherwise, agents in different partitions *must* have different partition id.

Scheduler encodes the runtime (an algorithm) of the distributed system. It is a tuple $\langle c, \zeta_t, \zeta_c \rangle$ where ζ_t is the task queue (a queue of actions), ζ_c is the consume queue (queue of statements), and c is a logical clock (lamport clock [20]) that is incremented by one each time a statement is executed. The role of a scheduler is to explore a sub-set of the allowed behaviors by the operational semantics in order to reveal a conclusion about the system under analysis.

5.2 Communication and Synchronization

Our model uses a minimal set of four communication and synchronization primitives. Such a restriction does not limit the expressivity of the language, while at the same time facilitating understanding. These primitives are now explained. **Send** is a fire-and-forget type of statement. Its signature is $send(\alpha_{src}, \mu, \alpha_{dst})$. **Ask** is a fire and return a handle (future) statement. Its signature is $ask(\alpha_{src}, \mu, \alpha_{dst}, vn)$. Its details are similar to that of send, except for vn which is a *variable name* where the handle (future) f is stored in source agent α_{src} local state. When an ask statement is fired, the message μ is sent to the destination agent, by a *temporary agent* on behalf of the source agent α_{src}. Later in time, there may/not be a reply (possibly from another agent than α_{dst}) to the sender of the message that resolves the future when received by the temporary agent. The temporary agent would update the source agent α_{src} upon receiving that reply. **Get** is the statement used to *block on* a future object. Blocking means not executing any more statements till that future object is resolved, then the agent is unblocked to schedule, consume, and/or execute tasks/statements. There are *two* variants of get statements. The blocking-get signature is $get(\alpha, vn, vn2)$ and the timed-get signature is $get(\alpha, vn, vn2, to)$. The agent calling it is α, the variable holding the future is vn, the variable that holds resolved value is $vn2$, and the time out limit is to.

5.3 Predicates and Conventions

We first need to state our conventions. **References and Types** are inferred directly from alphabets $\alpha, \mu, \gamma, s, f, vn$ and $where$ as follows: an agent with $\alpha \in \mathscr{A}$, message with $\mu \in \mathscr{M}$, action $\gamma \in \Gamma$, statement $s \in Statement$, and future with $f \in \mathscr{F}$. We also refer to variable names with $\{vn, where\} \in Ids$, appended with a number if more than one. The same thing goes to other types. **Subscripts** are used in two ways. First, in parameters to indicate the function of the parameter e.g. α_s for source agent. Second, we use it to indicate where the entity belongs e.g. $\mathscr{L}_\alpha$ for local state of agent α. **Specific task in a scheduler** $\gamma_{\Sigma,\alpha}$ means a front-most action in a scheduler's task queue ζ_t that was scheduled by agent α. We also use the same **structure (tuple) as a predicate** with commas indicating place inside the tuple representing them as opposed to dots that are used to state the flexibility of location inside a tuple/sequence. `Involved(`s`)` returns a variable length tuple according to the kind of current statement. The tuple represents the arguments *involved* in this statement.

$$Involved(s) = \begin{cases} \langle \alpha_{src}, \mu, \alpha_{dst} \rangle & \text{if } s[k] = SEND \\ \langle \alpha_{src}, \mu, \alpha_{dst}, vn \rangle & \text{if } s[k] = ASK \\ \langle \alpha, vn, vn2 \rangle & \text{if } s[k] = GET \\ \langle \alpha, vn, vn2, to \rangle & \text{if } s[k] = TGET \\ undefined & otherwise \end{cases}$$

The meanings of symbols inside each tuple returned are as explained in Sect. 5.2.

5.4 Rules Walkthrough

Now, we explain the rules in Fig. 2 one by one, and illustrate the effects of these rules with the aid of the subfigures of Fig. 1 (when applicable). Let us start by explaining one rule, namely `SCHEDULE`.

Schedule. The `SCHEDULE` rule states that to schedule a task from an agent α, the agent needs to be in the unblocked ($\neg b$) state. Further, the agent should have an entry in its reactions map $\mathscr{R}$ that maps the message received, μ, residing at the head of its queue q (indicated by $\mu.q$), to an action. Moreover, the action field formal parameters m and a must be set to μ and α, respectively. Further, the action's execute queue ζ_x must refer to *stmts* (indicated by $\zeta_{x,\gamma} = stmts$), where *stmts* is nonempty. In addition, the `ChoseSchedule`(Σ, α) predicate must be true. The predicate means the scheduler *chose to schedule* a task from agent α. We use the "as" notation, as in Ocaml, to serve as an alias; for instance, γ_a *as* $\langle \mu, \alpha, stmts, \zeta_x \rangle$ uses γ_a as an alias for $\langle \mu, \alpha, stmts, \zeta_x \rangle$. The system transitions to a state where the message μ is removed from the front of agent α's queue, and the action γ_α is appended to the scheduler's task queue (ζ_t). An example of this rule before it triggered twice is shown in Fig. 1b and after it triggered in Fig. 1c.

Consume. The `CONSUME` rule fires when the scheduler (Σ) has a task in its task queue (ζ_t). In addition, that task is scheduled by a currently non-blocked ($\neg b$) agent α. Further, the predicate `ChoseConsume` (Σ, α) must return true, which means that the scheduler *chose to consume* from a front-most task (symbolized by $\gamma_{\Sigma,\alpha}$) that was scheduled by agent α. Then, the system transitions by (1) popping the statement that is at the front of the task's execute queue ($s.\zeta_x$) (2) appending that statement to the back of the scheduler's consume queue ($\zeta_c.s$). An example of this rule before triggering four times is shown in Fig. 1c and after it triggered in Fig. 1d.

Send. This rule (`SEND`) states that if the current statement s at the front of the scheduler consume queue ζ_c is a send statement (`IsSnd`(s)), and that statement parameters returned by `Involved(s)` are $\langle \alpha_s, \mu, \alpha_d \rangle$, and the statement is not a resolving send ($\neg IsRSend(\mathscr{T}, s)$), i.e. the destination α_d isn't a member of the temporary agents. In addition, neither the source agent α_s is blocked ($\neg b$) nor the destination agent α_d is locked nor they reside in different network partitions ($p = p'$). Then, if the scheduler *chose to execute* a statement (`ChoseExOne`(Σ)), the transition happens. That is, the message μ is appended to agent α_d queue ($q.\mu$). In addition, the statement is removed from the front of the scheduler's consume queue (ζ_c). An example of this rule when it triggers is shown in Fig. 1f and after it completes is shown in Fig. 1g.

Ask. `ASK` rule states that if the current statement to execute is an ask, predicate `IsAsk`(s), and like in `SEND` rule, the source agent is in unblocked state ($\neg b$) and the destination agent α_d is in unlocked state ($\neg l$), and they reside in the same network partition ($p = p'$). Then, if the scheduler *chose to execute* a statement, a fresh temporary agent α_t (i.e. $\alpha_{\mathtt{t}}$ =`fresh(Agent)`) is created along with a fresh future $f \in \mathscr{F}$ (i.e. `f` =`fresh(Future)`). Then, the transition happens: (1) the temporary agent α_t is added to the temporary agents $\mathscr{T}$, inheriting the same network partition as the asker/source agent (p'), (2) the future f is added to both the temporary agent and the source agent α_s local states under the key vn, i.e. $\mathscr{L} \cup (vn, f)$ (3) the temporary agent local state updated with the $(where, \alpha_s)$, to keep track *where to* forward the RF (Resolve Future) message in case the future was resolved. (4) the sender field of the message updated to be the temporary, $\mu(\alpha_t,)$ (5) the ask message enqueued at the destination agent α_d's receive queue, $q.\mu$. An example operation of this rule is visualized in multiple frames of Fig. 1, namely in frames 1e, 1f, and 1g. Important to notice that the updates to local states of both temporary and source agent stated here are *not* shown in the figure due to space constraints.

Resolving Send. `R-SEND` rule states the same guards as a regular `SEND` rule except that the destination is a temporary agent, i.e. predicate $IsRSnd(\mathscr{T}, s)$. As such, additional work need be done over a normal send by α_t. So, if the scheduler *chose to execute* the statement, the transition happens: (1) The temporary agent updates its future resolved status to `true` and that future's value from the first payload of the message μ, and encapsulates it in a fresh RF message setting its sender to the original sender α_s, i.e. $RF(\alpha_s, f(true, \mu[p_0]))$ (2) The RF message is then *inserted* into the destination agent α_d queue with the resolved future, before all non-Resolve-Future messages but after all other RF messages, as shown by $RF^*.RF(\alpha_s, f(true, \mu[p_0])).q$ (3) The temporary removes itself from the temporary agent's set, $\mathscr{T} \setminus \{\alpha_t(, , , , , \mathscr{L} \cup \{(vn, f(true, \mu[p_0])), (where, \alpha_d)\}, p')\}$. Up to this point, the future is considered resolved, however it is up to the scheduler implemented to decide when to update α_d local state with the resolved future, as can be told from the post state of the destination agent local state, $\mathscr{L} \cup (vn, f)$. An example of a resolving send executing is shown in multiple frames in Fig. 1, namely frames: 1k, 1l, 1m and 1n.

Resolved Get/Timed-Get. `R-GET` rule states that if the current statement is either a blocking-get (i.e. `IsGet(s)`) or a timed-get (i.e. `IsTGet(s)`), and the future they try to retrieve is already resolved,$f(true, val)$. In addition, that future is stored in α's local state under entry vn. Further, the scheduler *chose to execute* a statement. Then, that future's value is retrieved and stored in another entry in the local state of the same agent, i.e. $\mathscr{L} \cup \{(vn, f(true, val)), (vn2, val_f)$. Figure 1p shows the state before this rule triggered, and Fig. 1q shows the effect after it is triggered by agent c.

Blocking Get. `BLK-GET` rule states the same guards stated by `R-GET` except that: (1) the future in this case is *not* resolved (2) the current statement is a blocking-get ($IsGet$) (3) and the future is unresolved $f(false,)$. If the scheduler *chose to execute* a statement, a transition happens: All statements indicated by ss, that are returned by the $PreEmpted(\Sigma)$ in the same order they were consumed, are *appended* to the current statement and the resulting sequence of statements *prepended* to the front-most task execute queue, i.e. $s.ss.\zeta_x$. `PreEmpted`(Σ) determines all those statements, that were consumed from the same agent α tasks and returns them. Then, these statements in ss are removed from the consume queue, as in $\zeta_c \setminus ss$. Lastly, the agent blocking status is updated from unblocked $\neg b$ to blocked b. An Example of this rule prior it triggers is shown in Fig. 1i and after it triggers in Fig. 1j.

Timed Get. `T-GET` rule states the same guards as in `BLK-GET` rule except it does not block indefinitely. It only blocks temporarily until it times out ($to \leq 0$), delaying execution of all those statements till the agent unblocks. That is the time for them to have been consumed, i.e. appended to the consume queue of the scheduler, $\zeta_c.ss$ (skipping the statement s). Agent α's state changes to unblocking.

6 Faults and Exploration Semantics

We present the semantics for faulty behaviors that can, in our framework, be introduced during testing. These rules can be thought of as facilitating high-level fault injection. Structuring high-level fault injection rules in this manner ensure that users can cover faulty scenarios systematically.

In Sect. 6.1 we will walk through the rules shown in Fig. 3 and explain in detail how they work. The next section, Sect. 6.2, we will show how they can be used to simulate faults in the context of previous sections example, and other complementary examples when needed.

6.1 Rules Walk-Through

In order for a scheduler to determine the existence of a bug in the model, it needs some facilities. These facilities, as we mentioned before, include the presence of a locking mechanism to simulate network partitioning, fault injections to simulate, for example, message duplication, and many others such as network message delivery/dropping mechanisms like the incoming queue. The reader is highly encouraged to refer frequently to Fig. 3 in order to understand rules while reading this walk through.

Message Dropping. `MSG-DROP` rule states that if the scheduler (Σ) chose to drop a message (μ_m) from an agent ($\alpha(\ldots \mu_m \ldots, , , , , ,)$), shown as $ChoseDrop(\Sigma, \alpha, \mu_m)$, then that message is discarded from that agent's queue ($\alpha(\ldots, , , , , ,)$). This rule is of extreme importance, and the reason is that message dropping is the root cause of the majority of problems in

$$\text{MSG-DROP}\ \frac{ChoseDrop(\Sigma,\alpha,\mu_m)}{\langle\mathscr{A}(\alpha(\ldots\mu_m\ldots,,,,,,,)),\mathscr{T},\Sigma\rangle\rightarrow\langle\mathscr{A}(\alpha(\ldots,,,,,,,)),\mathscr{T},\Sigma\rangle}$$

$$\text{IMPL-DROP}\ \frac{\begin{gathered}[((IsSnd(s)\vee IsRSnd(s))\wedge\langle\alpha_s,\mu,\alpha_d\rangle=Invovled(s))\vee\\(IsAsk(s)\wedge\langle\alpha_s,\mu,\alpha_d,\rangle=Involved(s))]\wedge(p\neq p')\wedge ChoseExOne(\Sigma)\end{gathered}}{\begin{gathered}\langle\mathscr{A}(\alpha_s(,,\neg b,,,,p),\alpha_d(q,,\neg b,,,,p')),\mathscr{T},\Sigma(,,s.\zeta_c)\rangle\rightarrow\\\langle\mathscr{A}(\alpha_s(,,\neg b,,,,p),\alpha_d(q,,\neg b,,,,p')),\mathscr{T},\Sigma(,,\zeta_c)\rangle\end{gathered}}$$

$$\text{MSG-REORD}\ \frac{ChoseReOrder(\Sigma,\alpha,O)\wedge O\in\{P2P,\neg P2P\}\wedge QPermuted(q,O,q')}{\langle\mathscr{A}(\alpha(q,,,,,,)),\mathscr{T},\Sigma\rangle\rightarrow\langle\mathscr{A}(\alpha(q',,,,,,)),\mathscr{T},\Sigma\rangle}$$

$$\text{TASK-INTRLV}\ \frac{ChoseInterleave(\Sigma,O)\wedge O\in\{PO,\neg PO\}\wedge TPermuted(\zeta_t,O,\zeta_t')}{\langle\mathscr{A},\mathscr{T},\Sigma(,\zeta_t,\zeta_c)\rangle\rightarrow\langle\mathscr{A},\mathscr{T},\Sigma(,\zeta_t',\zeta_c)\rangle}$$

$$\text{MSG-DUPL}\ \frac{ChoseDupl(\Sigma,\alpha,\mu_m)}{\langle\mathscr{A}(\alpha(\ldots\mu_m\ldots,,,,,,,)),\mathscr{T},\Sigma\rangle\rightarrow\langle\mathscr{A}(\alpha(\ldots\mu_m\ldots\mu_m\ldots,,,,,,,)),\mathscr{T},\Sigma\rangle}$$

$$\text{NET-PART}\ \frac{ChosePartition(\Sigma,\{\alpha_{a0},\ldots,\alpha_{am}\})\wedge p'=fresh(Ids)}{\begin{gathered}\langle\mathscr{A}(\alpha_{a0}(,,,,,,p),\ldots,\alpha_{am}(,,,,,,p),\alpha_{b0}(,,,,,,p),\ldots,\alpha_{bn}(,,,,,,p)),\\\mathscr{T},\Sigma\rangle\rightarrow\\\langle\mathscr{A}(\alpha_{a0}(,,,,,,p'),\ldots,\alpha_{am}(,,,,,,p'),\alpha_{b0}(,,,,,,p),\ldots,\alpha_{bn}(,,,,,,p)),\\\mathscr{T},\Sigma\rangle\end{gathered}}$$

$$\text{NET-UNPART}\ \frac{\begin{gathered}ChoseUnPartition(\Sigma,\{\alpha_{a0},\ldots,\alpha_{am}\},\{\alpha_{b0},\ldots,\alpha_{bn}\})\wedge\\(p'\neq p)\wedge p''=fresh(Ids)\end{gathered}}{\begin{gathered}\langle\mathscr{A}(\alpha_{a0}(,,,,,,p'),\ldots,\alpha_{am}(,,,,,,p'),\alpha_{b0}(,,,,,,p),\ldots,\\\alpha_{bn}(,,,,,,p)),\mathscr{T},\Sigma\rangle\rightarrow\\\langle\mathscr{A}(\alpha_{a0}(,,,,,,p''),\ldots,\alpha_{am}(,,,,,,p''),\alpha_{b0}(,,,,,,p''),\ldots,\\\alpha_{bn}(,,,,,,p'')),\mathscr{T},\Sigma\rangle\end{gathered}}$$

$$\text{PROC-CRSH}\ \frac{ChoseCrash(\Sigma,\alpha)}{\langle\mathscr{A}(\ldots,\alpha,\ldots),\mathscr{T},\Sigma\rangle\rightarrow\langle\mathscr{A}(\ldots),\mathscr{T},\Sigma\rangle}$$

Fig. 3. DS2 faults semantics

distributed systems. As a matter of fact, the majority of faults can be reduced to message dropping or are a product of dealing with dropped messages. Our *non-primitive* network partitioning rules are built around *implicit message dropping* by communication rules (`SEND`, `ASK`, and `R-SEND`). When these three rules detect a different partition indicator/identifier between the source and destination agents, they drop the message to be sent; that is what rule `IMPL-DROP` in Fig. 3 does and is explained next.

Implicit Message Dropping. The rule `IMPL-DROP` states that on executing any statement ($ChoseExOne(s)$) that is either a send or a resolving send ($(IsSnd(s) \vee IsRSnd(s))$), or an ask ($IsAsk(s)$) whose involved communicating agents (a_s and a_d), shown as $\langle\alpha_s,\mu,\alpha_d\rangle = Involved(s)$ for send/resolving-send and as $\langle \alpha_s, \mu, \alpha_d, \rangle = \mathit{Involved}(s)$ for ask, reside on different network partitions ($p \neq p'$) then the message (μ) is implicitly dropped. Hence, the post state is the exact equivalent to the prior state of the distributed system.

Message Reordering. `MSG-REORD` rule states that if: (1) The scheduler (Σ) chose to reorder ($\mathit{ChoseReOrder}(\Sigma, \alpha, O)$) an agent's ($\alpha$) incoming queue according to one of the policies specified by $O \in \{P2P, \neg P2P\}$ (`P2P` policy means that the new reordering of messages must keep the relative order between point-to-point communication, i.e. messages sent from the same sender and to the same receiver should maintain their relative order), and (2) the effect of permuting these messages in the old version of the queue (q) according to the policy specified above, $\mathit{QPermuted}(q, O, q')$, resulted in a policy-respecting queue of messages (q'), then the system transitions and the new reordering of messages takes effect.

Task Interleaving. Task interleaving rule `TASK-INTERLV` states that if a scheduler (Σ) chose to interleave its task queue (ζ_t), stated as ($\mathit{ChoseInterleave}$ (Σ, O)), according to one of the polices $O \in \{PO, \neg PO\}$, then the resulting queue (ζ_t') is a permutation of the old task queue that conforms to the policy specified ($\mathit{TPermuted}(\zeta_t, O, \zeta_t')$). The policy PO means that the interleaving of tasks should respect "program order". That is, the order of tasks scheduled by the same agent should stay the same relative to each other.

Message Duplication. The message duplication rule (`MSG-DUPL`) is a fault injection mechanism. This rule states that if a scheduler (Σ) chose to duplicate a message (μ_m) that is in an agent's queue ($\alpha(\ldots \mu_m \ldots, , , , , ,)$) – all of that is stated as $\mathit{ChoseDupl}(\Sigma, \alpha, \mu_m)$ – then it inserts a copy of it anywhere in the agent's queue ($\alpha(\ldots \mu_m \ldots \mu_m \ldots, , , , , ,)$) in a way that is specific to the target task of that scheduler. This is why a certain position of where the duplicated message is to be inserted into an agent's queue is not specified. Other kinds of message duplication can be due to either crashes of processes followed by retries e.g. Re-transmission of such messages till an ack is received, or due to some other *behavior induced* re-transmission[3]. Such behaviors are not controlled by our model, since they are specified by target systems' developer(s), and do impose causality constraints for message duplication. However, they are explorable by schedulers even if said schedulers do not use fault injection mechanisms. Fault injection using message duplication is a way to give that last nudge (when appropriate) to the system to hit a bug, and hence is prioritized last compared to other rules that simulate faultiness.

Network Partitioning. There are two rules that manipulate network partitioning (`NET-PART` and `NET-UNPART`), and another that makes use of network partitioning to implicitly drop messages (`IMPL-DROP` explained before). The network partition rule `NET-PART` states that if the scheduler (Σ) decided to partition a set of agents ($\{\alpha_{a0}, \ldots, \alpha_{am}\}$) away from a distributed system – that is to place them in a separate partition of their own – then the partition identifier (p) in these agents is changed to a fresh partition identifier (p'), indicated

[3] e.g. a heart beat sending messages every certain time period.

by $p' = fresh(Ids)$. The network unpartitioning rule `NET-UNPART`, on the other hand, (1) takes two sets of agents having different partition numbers per set, (2) creates a new partition id for the resultant combined partition, and (3) updates the partition id on all of the agents from both sets/partitions to be (p''), i.e. the new partition id. After which, all agents reside in the same partition (p'') and can communicate with each other. An implementation of the model may choose any function that guarantees the deterministic resultant partition id for combining the two. The model does not impose any restriction to do that or not.

Process Crashing. The process crashing rule `PROC-CRSH` states that if a scheduler (Σ) chose to crash ($ChoseCrash(\Sigma, \alpha)$) an agent ($\alpha$), then that agent simply ceases to exist in the distributed system ($\mathscr{A}(\ldots)$), along with its stored state (queue and local state[4]).

In the next section, we will discuss how these fault rules make sense in a realistic system exploration by discussing the previous example and introducing high level complementary ones when needed.

6.2 How Faults-Rules Help

From Fig. 1 we will try to focus on deadlock detection to illustrate the importance of the rules shown in Fig. 3. This helps in illustrating the benefits of the faults we can model to address realistic situations. In the case that the deadlock detection example does not apply to illustrate a rule, we will introduce a complementary example to help.

Message Dropping. A message is not considered delivered till it is handled by the agent, e.g. by stashing it or by processing (potentially producing more messages, and other possibly irreversible side effects). This leaves sufficient room for the model to address realistic message dropping scenarios to model different faults. One example is that we can model packet/message loss by dropping that message from the destination agent's queue. We could have dropped the resolve future message (RF) from the client's queue to simulate the packet/message loss. The client, then, can deadlock and the scheduler will be able to detect that. These examples (along with upcoming network partitioning examples) show the importance of the ability to simulate message dropping in a formal model for distributed systems.

Message Reordering. An intuitive example of message reordering rule benefit is easily shown in multiple examples. One could think of a bank account getting mutated by two clients at the same time, $c1$ and $c2$. These clients issue their orders independently, while the bank account keeps track of what balance remains to be withdrawn. Let us assume that the initial balance is zero and it is updated in real time, i.e. no human supervision and/or review is involved in updating the balance. Client $c1$ deposits some money, but meanwhile $c2$ is withdrawing from the balance. If $c1$'s message/update reaches before $c2$ request

[4] For this base model being discussed in this work, all it has as a state is a queue and a local-state. Implementations, of course, can have more than that.

is received and fulfilled, $c2$ is happy. If, however, $c2$'s request happens before $c1$'s request, $c2$ is not happy. Client $c2$ keeps on re-issuing the same request, but withdrawal is always prioritized (or reaches the server faster) over $c1$'s. There, we can see that message reordering simulating delayed messages (relative order of messages arriving at the server) taking bad effect on the outcome of $c2$. The same scenario happens with updating any shared resource that supports any non-commutative pair of operations, except it may go much deeper than two interactions between just three agents. All of these scenarios are enabled by the message reordering rule.

Message Duplication. Consider the same example for the bank account discussed above. Since $c2$ is duplicating the request often, there could be the possibility of repeating a withdrawal. It could be the case that one request took completely different route to the server, and got delayed beyond anticipation. Due to that, $c2$ times out and normally re-issues the same request. However, the old message gets delivered along with the new one and both get processed. Of course, this duplication is very common in distributed systems, and normally processes should be coded to handle such scenarios. It is helpful to have such exploration ability in the model to check for unwanted behavior due to message duplication and that logic handling such duplication handles the situation correctly.

Tasks Interleaving. In the case that message reordering is short of showing a bug, can there be a tweaked task interleaving to target that bug and show its existence? It turns out that, in our bank account example, if the server dispatches its tasks in a program-order (PO) and point-2-point ($P2P$) respecting manner, there still could be a way to interleave the tasks of two different agents in a way that they cause problems, e.g. overdraft. The worst case scenario would be the server dispatches all tasks coming from $c2$ then the account balance is doomed, if there is no overdraft protection. If at least one $c2$ task was dispatched before $c1$ deposits money, a bit more optimistic, it still can cause overdraft. A scheduler relying on this rule can explore said scenarios, even if message reordering fell short of revealing a bug.

Network Partitioning. One example is that we can model a network partition between the client and the server shown in the example, dropping all messages going either way implicitly by any communication primitive described previously. Another example is to simulate a network partition before that RF message (in example shown in Fig. 1) is sent to the client and it will be auto-dropped based on the partitioning semantics and the `IMPL-DROP` rule semantics. The same network partition, between the client and server, can actually occur before the client sends its request. That would lead to the request to be lost (i.e. not delivered to the server for handling), and then the way the client is coded assumes it was delivered. That, in turn, will lead the client to *block* over a promised future whose request isn't even delivered, leading to another deadlock scenario. A more involved network partitioning scheme that may be simulated by network partitioning (implicitly message dropping between partitions) is to keep two partitioned sets of agents to evolve their state and stay divergent to simulate some consistency violation scenarios.

Process Stop/Crash. Going back to our deadlock detection example, from Fig. 1, it is easy to crash the server after/before the client request is issued (but before the server processes it) to cause a deadlock at the client. That server may come back online, after it lost the message, or it can stay down forever, leaving the client blocking on an unresolved future. Under the assumption of having some kind of a persistent state private to each agent, e.g. by checkpointing in-memory data to the disk, there would be an interesting set of examples to investigate the state consistency between processes that occasionally crash then reboot and weather they may recover and converge from state view divergence. Our implementation of the model takes advantage of the assumption that there is always private disk (persistent) store available to each agent, in order to be able to simulate that. Actually, in any distributed system, when there is an agent that is to process requests on which other agents are waiting or that causes more updates to propagate through the distributed system, crashing certain processes is a good idea to cause data consistency problems across the system (divergent state view) and/or other concurrency problems (e.g. deadlocks). After all, a lot of distributed systems problems/faults are due to message dropping, processes demise and/or reboots.

7 Bug Discovery in Snapshot of Runtime

By forcing us to think clearly, the operational semantics helped expose a bug in the original implementation of snapshotting support. For example, before we had an operational semantics, we struggled to correctly capture snapshots of the runtime state (which includes both the distributed system and the scheduler attached to it). For the most part, a system and its scheduler must be "deep copied," to create an isomorphic state; however, not everything can be copied verbatim. When copying a distributed system, the steps must be carried out in two phases: (1) a copy phase (creating objects but leaving references untouched) for all entities in it, followed by (2) a link phase (*re-wiring* references to entities from new snapshot) for all of them. However, in our original implementation, this was not the case. For example, an action's ζ_x statements kept on referring and affecting the original distributed system's agents even after the linking phase. More over, ζ_x statements were not the same statements from the snapshot's `stmts` template. So, when the link operation updated the agent field `a`, which in turn updates all of the template statements, it did not reflect in those inside of ζ_x. The operational semantics exposed the bug in the original snapshotting implementation that left actions in the snapshot *still attached* to the parent distributed system instead of their snapshotted counterparts. The operational semantics were essential in correcting snapshotting, which will be the cornerstone of DS2's model checking and testing functionality.

After correcting this mistake, DS2 can now easily and reliably capture, fork, and restore distributed system states. Listing 1.3 shows how succinctly one can now snapshot a whole distributed system along with its scheduler state and then restore from it.

```
val state = saveState(sch)
restoreState(sch,state)
```

Listing 1.3: Capturing and restoring to runtime state

A key detail here is that the scheduler `sch` *need not be the same scheduler that saved the state*. This flexibility enables different schedulers, for example, those running truly concurrently to explore different paths in the distributed system's state-space; and/or cooperation between different analyzing schedulers, i.e. switching on the fly between different kinds of analyses to be performed by different schedulers. An additional importance of this snapshot feature is enabling backtracking/stateful algorithms.

8 Related Work

Several distributed-system related projects have emphasized the use of formal semantics. SimgGrid [7] focuses on the simulation and model checking of distributed systems. In Verdi [29], the Coq system is used to develop formal operational semantics for network and node-failure models to synthesize distributed systems from specifications. Our work focuses on targeted correctness of distributed systems, as opposed to performance simulation as in SimGrid. Our goal is to allow designers to model a variety of distributed systems in the DS2 language; the formal definition of a modeling language is not targeted in Verdi.

MoDist [30] is a transparent operating system-agnostic model checker for unmodified distributed systems. Our work is extensible to address specific needs of distributed systems using custom schedulers. MaceMC is an execution based model checker for distributed systems specific to the Mace language. It benefits from coarse-grained interleaving exploration and random walk [17].

IronFleet [13] is a layered approach to TLA-style [22] state-machine refinement and Hoare-logic [15] based verification to synthesize provably correct distributed systems, developed by Microsoft Research.

Thanks to its utilization of operational semantics information for control flow statements and state space reduction policies, SAMC [23] was able to show huge gains, both in performance and precision, over regular Dynamic Partial Order Reduction (DPOR) based approaches.

Pony [2] is an actor based programming language that relies on avoiding blocking (i.e. lockless) to produce high performance distributed systems. Future support for fault-tolerance is expected to be based on Erlang's actors supervisory hierarchies [8].

Distributed Closures [10] rely on fixing distributed data (i.e. not allowing mutation) that are called "silos" and sending closures (function shipping) that are called "spores" instead of sending messages. These functions construct a lazy graph of computations over these constant distributed data that are finally evaluated/computed when needed to be materialized. This provides strong type safety, a functional approach to programming distributed systems, and some fault tolerance.

An approach for sequential programming (actors) in distributed systems [19] proposes a paradigm shift in programming distributed systems, by suggesting programming language level support to address said systems. Future direction of this work may support fault tolerance.

9 Conclusion

The current implementation status of our DS2 core is in its final stages of testing, and tightly follows the formal operational semantics explained in this work. We are working on developing an extended version of linearizability checking scheduler for distributed systems inspired by Line-up [6] and guided by the operational semantics explained in this work. Our immediate targets are developing a front-end parsing the benchmarks we developed in Akka: Paxos [21], Chord [26], and Zab [18] for our linearizability checker, semantics-aware distributed systems automated testing, analysis, and synthesis. We are developing many use-cases [4,9] to drive our work forward along these lines.

Acknowledgments. We acknowledge NSF grant CCF 7298529, and help by Heath French, Jarkko Savela, Zepeng Zhao, and Anushree Singh. This work is, also, supported in part by NSF Award CCF 1346756.

References

1. Akka, February 2016. http://akka.io/
2. Pony - High Performance Actor Programming, November 2016. http://www.ponylang.org/
3. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
4. Al-Mahfoudh, M., Gopalakrishnan, G., Stutsman, R.: Toward rigorous design of domain-specific distributed systems. In: Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering, FormaliSE 2016, pp. 42–48. ACM, New York (2016)
5. Burckhardt, S.: Principles of eventual consistency. Found. Trends Program. Lang. **1**(1–2), 1–150 (2014)
6. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. SIGPLAN Not. **45**(6), 330–340 (2010)
7. Casanova, H.: SimGrid: a toolkit for the simulation of application scheduling. In: Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 430–437 (2001)
8. Clebsch, S., Pony: co-designing a type system and a runtime. SPLASH 2016. ACM, New York, November 2016. Appeared in SPLASH-I, Presentation
9. DS2 official website (2016). http://formalverification.cs.utah.edu/ds2. Accessed 31 Jan 2016

10. Haller, P., Miller, H.: Distributed programming via safe closure passing. In: Gay, S., Alglave, J. (eds.) Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18 April 2015. EPTCS, vol. 203, pp. 99–107 (2015)
11. Haller, P., Odersky, M.: Event-based programming without inversion of control. In: Proceedings of the 7th Joint Conference on Modular Programming Languages, JMLC 2006, pp. 4–22. Springer, Heidelberg (2006)
12. Havelund, K., Pressburger, T.: Model checking java programs using java pathfinder. Int. J. Softw. Tools Technol. Transf. **2**(4), 366–381 (2000)
13. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, 4–7 October 2015, pp. 1–17 (2015)
14. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI 1973, pp. 235–245. Morgan Kaufmann Publishers Inc., San Francisco (1973)
15. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
16. Holzmann, G.J.: Logic verification of ANSI-C code with SPIN. In: SPIN. Lecture Notes in Computer Science, vol. 1885, pp. 131–147. Springer (2000)
17. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2009)
18. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN), pp. 245–256, June 2011
19. Kuraj, I., Jackson, D.: Exploring the role of sequential computation in distributed systems: motivating a programming paradigm shift. In: Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, pp. 145–164. ACM, New York (2016)
20. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
21. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
22. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co. Inc, Boston (2002)
23. Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J.F., Gunawi, H.S.: SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI 2014, pp. 399–414. USENIX Association, Berkeley (2014)
24. McCaffrey, C.: The verification of a distributed system. ACM Queue **13**(9), 60:150–60:160 (2015)
25. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC 2014, pp. 305–320. USENIX Association, Berkeley (2014)
26. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols

for Computer Communications, SIGCOMM 2001, pp. 149–160. ACM, New York (2001)

27. Strategy design pattern (2017). https://sourcemaking.com/design_patterns/strategy. Accessed 1 Jan 2017
28. Tasharofi, S., Karmani, R.K., Lauterburg, S., Legay, A., Marinov, D., Agha, G.: TransDPOR: a novel dynamic partial-order reduction technique for testing actor programs. In: Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems, FMOODS 2012/FORTE 2012, pp. 219–234. Springer, Heidelberg (2012)
29. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. ACM SIGPLAN Not. **50**(6), 357–368 (2015)
30. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: transparent model checking of unmodified distributed systems. In: NSDI, pp. 213–228. USENIX Association (2009)
31. Zave, P.: How to make chord correct (using a stable base). Computing Research Repository (CoRR), abs/1502.06461 (2015)

BHive: Behavior-Driven Development Meets B-Method

John Douglas Carter(✉) and William Bennett Gardner

School of Computer Science, University of Guelph, Guelph, ON, Canada
{jcarter, gardnerw}@uoguelph.ca

Abstract. Behavior-Driven Development (BDD) is an "outside-in" approach to software development built upon semi-formal mediums for specifying the behavior of a system as it would be observed externally. Through the representation of a system as a collection of user stories and scenarios using BDD's notation, practitioners automate acceptance tests using examples of desired behavior for the envisioned system. A formal model created in concert with BDD tests would provide valuable insight into test validity and enhance the visibility of the problem domain. This work called BHive builds upon the formal underpinnings of BDD scenarios by mapping their "Given," "When," and "Then" statements to "Precondition," "Command," and "Postcondition" constructs as introduced by Floyd-Hoare logic. We posit that this mapping allows for a B-Method representation to be created and that such a model is useful for exploring system behavior and exposing gaps in requirements and test plans. In this extension of previous work, we outline recent additions to BDD tooling required for the described integration, present a new strategy for test case generation from our approach, and expand on the benefits of the BHive approach to integrating formalism within a BDD project.

Keywords: BDD · Behaviour-Driven development · B-Method · Agile · Test generation

1 Introduction

The term "Formal Methods" broadly describes approaches for specifying, modelling and verifying systems that are grounded in mathematical rigor. A system properly developed with such treatment can be proven to have (or not have) specific properties or behavior. Successful application of formal methods typically depends on contributions from "guru-level" experts. This specialized knowledge requirement and perceived costs (both time and money) often reserve the application of these methods to projects requiring higher levels of assurance than can be provided using empirical testing practices.

The rise in popularity and reported productivity of agile software development practices have given many development houses pause to evaluate the effectiveness of so-called "heavyweight" methods in use today, particularly the "waterfall" development process. Nonetheless, it seems that formal methods are typically applied to projects following a traditional, sequential development process. How can we account for the apparent "impedance mismatch" between formal and agile methods? A number of reasons will be put forward in Sect. 2. Despite these, we believe that a useful bridge can be built between the formal and the agile worlds. This work is about using the "hidden" formalism in one agile method, Behavior-Driven Development (BDD), to connect it with B-Method via the Python tooling for BDD known as Behave. We call this technique "BHive" and will illustrate it below using a small case study. The purpose of this paper is to describe the conceptual basis of BHive; actual software tools will be developed as future work.

This paper builds on previous work [1] about the BHive approach. Here we expand on that description, outline enhancements to the development process including our provided tool support, and describe initial work toward test case generation using BHive.

The next section will give background on BDD and B-Method. Section 3 introduces BHive and applies the technique to a "Take-A-Number" machine case study. Section 4 describes BHive's approaches to verification, and Sect. 5 the BHive development flow. Section 6 comments on the applicability of this approach. Related work targeting integration of formal and agile methods is presented in Sects. 7, and 8 gives the conclusion and future work.

2 Background

Agile methods are lightweight methodologies that strive to implement principles outlined in the Agile Manifesto [2]. Methods identified as Agile typically share the following characteristics:

- The ability to embrace change and "course correct" anywhere in the project lifecycle without incurring undue overhead.
- Cross-functional teams working in close collaboration, including project stakeholders external to the development team (i.e., customers, domain experts, business analysts).
- Continuous delivery of incremental milestones: By providing deliverables early and often, the clients of agile teams can see the product grow and can shape its development through clarification of requirements and feedback.
- An adaptive approach to analysis and design: "Big design up front" is replaced with deliveries of smaller groups of functionality. The business value of milestones determines their delivery priority. A tight, continuous feedback loop informs decisions and shapes future development and management.

We suggest a combination of reasons why agile and formal methods seem to make strange bedfellows:

- Software vendors working in markets that require formal methods-backed assurances tend toward the more structured, document-driven "traditional" approaches such as Waterfall or "Vee" models [3]. The development approach itself may be part of a larger industrial certification process required by prospective customers.
- A traditional sequential approach contains dedicated phases for analysis and design. These phases may appear as a better fit for formal model-building activities than the periodic delivery cadences used by agile.

- Methodology experts may not be available for the entire development process, or the cost to retain them throughout may be judged too high.
- Agile methods focus on incrementally delivering business value to the customer. Modelling the entire system before development could delay initial milestones. “Front-loading” specification and modelling could be seen to be at odds with the agile notion of evolutionary development—requirements changing alongside a shared understanding between developer and customer.
- Agile methods favor “individuals and interactions over processes and tools” and “working software over comprehensive documentation” [2]. A formal model could be relegated to a second class artifact or seen as unnecessary documentation by the customer.
- Proponents of the agile movement assert that the “old way” of developing software is broken, citing collateral damage such as schedules and budgets. Formal methods may be deemed an unnecessary cost of time or money.
- Agile methods are relatively new compared to both traditional development approaches such as Waterfall, and to formal methods themselves.
- Development teams working in some sectors (e.g.: mobile application development, web development, game development) may have been educated in a highly focused setting and have not been exposed to a breadth of software engineering methods, perhaps learning Agile “on the job.”

The subsections below introduce the building blocks of the BHive technique, BDD, and B-Method.

2.1 Behaviour-Driven and Acceptance Test Driven Development

BDD is an emerging agile method. In the originating work [4], Dan North describes how BDD began as a shift in the process of naming and thinking about writing unit tests. In applying Test-Driven Development (TDD), North observed that merely changing the name of a unit test resulted in a marked change in how developers thought about test cases and communicated their understanding of the system. Beginning the name of a unit test with the word “should” makes the test more of a sentence and summarily captures an external description of behavior desired from the class. North suggests that any inability to apply such a template implies the behavior captured by the test belongs elsewhere. BDD also incorporates one of the principles of Domain-Driven Design—the “Ubiquitous Language” (UL). Evans asserts [5] that a useful model should be syntactically structured around the “domain model” and that this model (and its associated terminology) become the primary means to connect all team members with the product being developed.

BDD has commonality with Acceptance Test Driven Development (ATDD), to the extent that occasionally the terms are used interchangeably. There is, however, an important distinction: ATDD builds on the “test first” philosophy of TDD but moves the focus from testing internal functional blocks to automated tests of acceptance criteria. ATDD captures automated acceptance tests in the language of the business domain; a passing test corresponds to one or more business goals being achieved by the system under development. In contrast, a BDD specification is a user story, not a test. A BDD story exemplifies the desired behavior of the envisioned system as viewed by an external observer. This vantage point, combined with a structured natural language medium, provides specifications that are meaningful and contextual to all project stakeholders while still allowing automated processing performed by a BDD tooling package.

Gherkin [6] is a "business readable" domain specific language used by many BDD frameworks: Cucumber for Ruby, Behave for Java, Behat for PHP and Behave [7] for Python. The relationship between BDD's components is illustrated in Fig. 1 below:

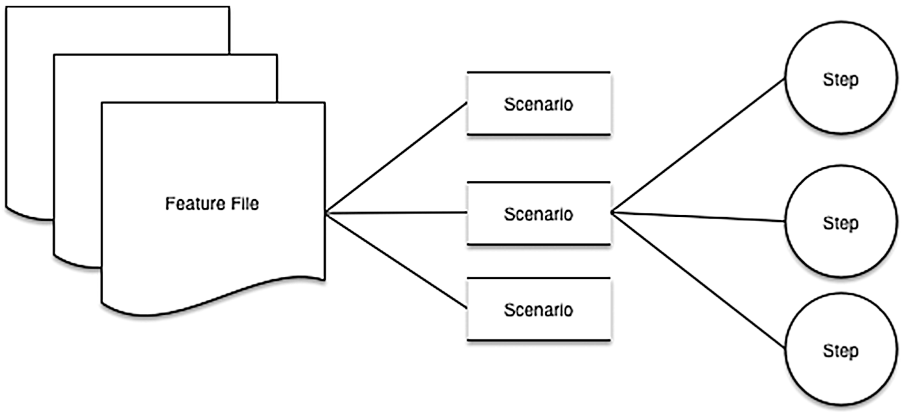


Fig. 1. Entity relationship diagram of BDD components

Thus, a system specified using Gherkin contains many "feature files." Each feature file collects "scenarios" that form an executable specification of its behavior. Scenarios are structured out of "steps" using a Given-When-Then (GWT) syntax:

```
Given: (a precondition)
When: (a triggering event)
Then: (a post condition)
```

The BDD tooling allows for linking each of the steps to a function, and for any number of features to be executed by the framework. During execution, the "Given" step should put the system-under-test into the state necessary for the "When" step to take place—which is where a state change occurs—and be observed by the "Then" step. Execution of a scenario can be parameterized using tables of data, allowing for real-world examples to be added alongside corresponding user stories.

We observe that the GWT construct is related to the concept of a "Hoare Triple" as introduced in Floyd-Hoare logic [8]. Within Floyd-Hoare logic Given-When-Then is represented as:

```
{P} C {Q}
```

where P is known as the "precondition", C the "command", and finally, Q the "postcondition". The triple and its accompanying axioms provide means to reason about program correctness for a system specified using the formalism. The Hoare triple is fundamental to modern state-based formalisms such as Z, B, and Event-B, discussed next.

2.2 B-Method

Invented by J.R. Abrial (the creator of Z) the B-method [9] provides the means to specify systems as “B Machines” using B’s “abstract machine notation” (AMN). Whereas Z focused purely on specification, B-method introduces additional granularity in its specification capabilities to enable refinement of specifications to an implementation. Machines contain events that change the system state (a machine’s variables). These events are composed of a name, parameters (a precondition that must be true for the event to be applied), and a state change. The machine itself provides definitions for constants, typing information for variables, and a set of invariants—logical propositions that must hold for correctness. Event-B [10] is an evolution of B-Method (often referred to as “classic B”).

By using B-Method or Event-B to model a system under development, a project team may perform correctness proofs on components of the system. Formalizing the system introduces precise logical semantics not found with design documentation or source code. Both B and Event-B provide support for model “refinement”—the gradual introduction of complexity into a successive version of the model. The equivalence between the refined model and the previous model is proven using “linking invariants.” With B-Method the end goal is the refinement to source code, whereas the Event-B refinement process is used to introduce model complexity gradually. The application of such a refinement process is akin to the agile concept of iterative development. Moreover, it allows for systems that are “correct-by-construction,” in contrast to implementing from a static up-front specification from which the system diverges over the course of the development lifecycle.

The specification process adds business value, in that the development team is forced to analyze and formally capture the behaviour of the system along with its data model, but also examine issues of system decomposition and the structure of system components. Rigid formal semantics allow for thorough understanding and verification of the system’s behavior. Such understanding is of particular importance in cases where system failure is beyond the level of inconvenience and even carries catastrophic consequences.

3 Introducing BHive

This work provides the means to bridge the expressive power of Behavior-Driven Development with the rigor of B-method. Development of this approach builds upon the Python BDD tooling package “Behave.” To form a semantic bridge between Gherkin and B-Method, BHive wraps a number of built-in Behave functions, and provides an alternate typing system as well as additional provisions not specific to Behave. BHive’s approach and integration are designed to augment an existing BDD methodology and its tooling. Modifications to syntax are avoided to ensure the approach is usable by existing teams applying BDD, and to minimize the effort needed to port BHive to other BDD toolings.

Figure 2 shows a high-level view of the inputs (feature files, step definitions, and environment file) and the output (i.e., the system itself) of a typical BDD flow using Behave. The dark grey arrows show flow into and out of Behave, and the curved arrow from output to input represents the iterative nature of the BDD process.

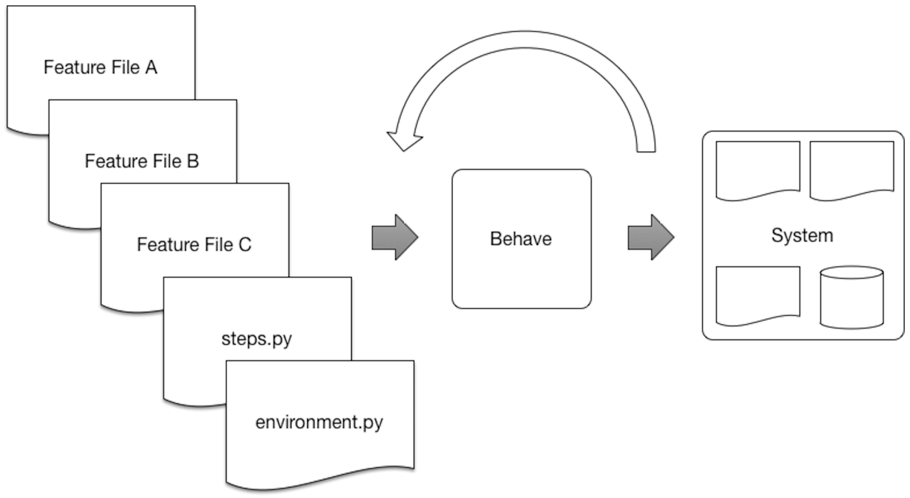


Fig. 2. Basic components of Behave

Figure 3 shows a view of BHive at the same level of generalization as Fig. 2. BHive accepts the same feature files, step specifications, and environment files that make use of BHive's integration functions, and produces B Machines represented in Abstract Machine Notation (AMN). In addition to the generation of B specifications, test cases are generated using the ProB tool.

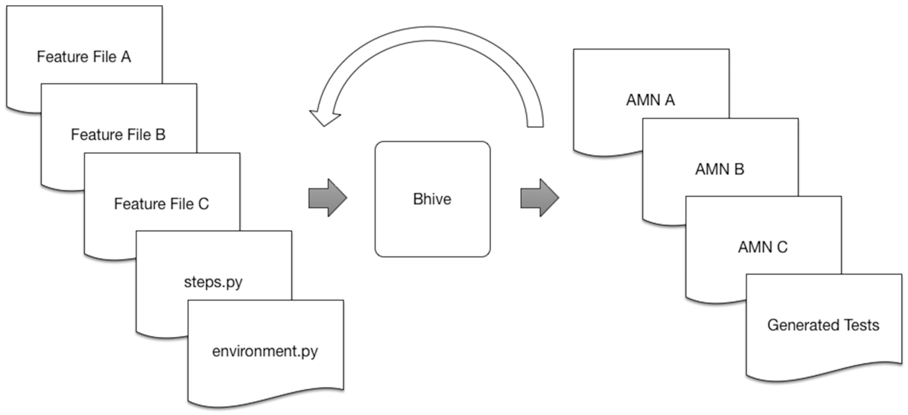


Fig. 3. Basic components of BHive

3.1 “Take-a-Number” Case Study

One of the case studies used to illustrate this work is a readily understandable “take-a-number” customer queuing system. The discussion begins with a feature file (in Gherkin) capturing the behavior of the system, followed by examples of how Gherkin and BDD are bridged.

Feature File. Here we are specifying part of the “TakeANumber” machine. The machine name is derived from the name of the feature file, in this case “TakeANumber.feature”.

```
Feature: Ticketing
  As a business owner, I want to effectively manage cus-
tomers waiting to be served in a "first come, first
served" manner.

  Scenario: Start_Machine
    Given Machine is off
    When  Machine is started
    Then  Machine is on
    And   Display shows 1
    And   Ticket is 1

  Scenario: Stop_Machine
    Given Machine is on
    When  Machine is stopped
    Then  Machine is off
    And   Display shows 0

  Scenario: Customer_Takes_Ticket
    Given Machine is on
    When  Customer takes <ticket>
    Then  Display should be less than or equal to the
ticket

  Scenario: Serve
    Given Machine is on
    And   Display is less than <ticket>
    When  A customer is called
    Then  Display should be less than or equal to the
ticket

  Scenario: Reset_Machine
    Given Machine is on
    When  Machine is reset
    Then  Display shows 1
    And   Ticket is 1
```

If we run Behave with this feature file, it will generate stubs for each of the steps we must create. The stubs generated for the "Start Machine" scenario are shown next:

```
@given(u'Machine is off')
def step_impl(context):
    raise NotImplementedError(u'STEP: Given Machine is
off')

@when(u'Machine is started')
def step_impl(context):
    raise NotImplementedError(u'STEP: When Machine is
started')

@then(u'Machine is on')
def step_impl(context):
    raise NotImplementedError(u'STEP: Then Machine is
on')

@then(u'Display shows 1')
def step_impl(context):
    raise NotImplementedError(u'STEP: Then Display shows
1')

@then(u'Ticket is 1')
def step_impl(context):
    raise NotImplementedError(u'STEP: Then Ticket is 1')
# ... stubs continue ...
```

These initial steps are identical whether one is following the Behave or the BHive approach. With Behave, development would likely continue with TDD or the "Test First" approach. The central tenet of TDD is that the first code one writes is a test, which is then followed by just enough code to make the test pass, and finally a possible refactoring step to improve or generalize the code one just wrote. Running the tests on demand allows one to evaluate the correctness of the refactor. Once satisfied, the process begins with a new unit test.

Part of writing "just enough code to make the test pass" is often the use of mocks. Mocks allow a developer to substitute another object, make calls on it, and enforce certain properties (parameters, return types) on the mocked object. This practice is also applied in BDD. Using mocks, the developer can implement the steps with objects that do not exist and make assertions on their behavior. It also allows a design to evolve, as the developer has free rein to code to the interface they want. It is this approach that links Behave's steps to a B-Method machine.

BHive provides a mocking library to specify Given, When, and Then steps. Unlike other mocking libraries which are free form, the BHive mocking library is slightly more restrictive in order to ease translation to B's Abstract Machine Notation.

In the Then step, we specify a test to verify that the change did take place. This will serve as input into the test generation process. Fleshing out the "Start Machine" scenario introduced above, we specify the following:

```
@given(u'Machine is off')
def step_impl(context):
    declare_variable(context, 'running', 'BOOL', 'FALSE')
    declare_variable(context, 'display', 'INT', '0')
    declare_variable(context, 'ticket', 'INT', '0')
    context.state.ensure_that('running','=','FALSE')
```

Here we specify the "Given" step. We declare three variables with their associated types and initial values. BOOL and INT types are B-Method built in types, but custom types may be introduced by the developer. The fourth parameter is the variable's initial value. This will be used for the INITIALISATION clause in the synthesized B Machine. The "ensure_that" method is used to specify conditions necessary to meet the behavior "Machine is off", in this case ensuring that the machine's "running" variable is false.

The Python variable "context" is Behave's mechanism for passing data between Gherkin constructs. It exists as an associated array with the following choice of scopes[1]: all, feature, and scenario. Whenever a new scope is entered (in this case the "Scenario" scope for "Start Machine") a new associative array is pushed on Behave's internal stack. This associative array exists for the execution of "Start Machine" and is popped from the stack after all of its steps have been executed.

BHive extends the context mechanism to perform "bookkeeping" on the state changes contained in a scenario. When a variable is declared, BHive associates it with the Feature being specified and allows for state changes to occur for that variable using the context.state.assign method shown below:

[1] A fourth scope, tag, does exist which provides a scope of each unique tag applied to a scenario. This feature is not presently used by BHive..

```
@when(u'Machine is started')
def step_impl(context):
    context.state.assign([('running', ':=', 'TRUE')])
    context.state.assign([('display', ':=', '1')])
    context.state.assign([('ticket', ':=', '1')])
```

Above, the “When” step is specified as the state change to satisfy “Machine is started”: running is enabled, and display and ticket are set to 1.

```
@then(u'Machine is on')
def step_impl(context):
    context.state.test_that(context, 'running', '=',
'TRUE')

@then(u'Display shows 1')
def step_impl(context):
    context.state.test_that(context, 'display', '=', 1)

@then(u'Ticket is 1')
def step_impl(context):
    context.state.test_that(context, 'ticket', '=', 1)
```

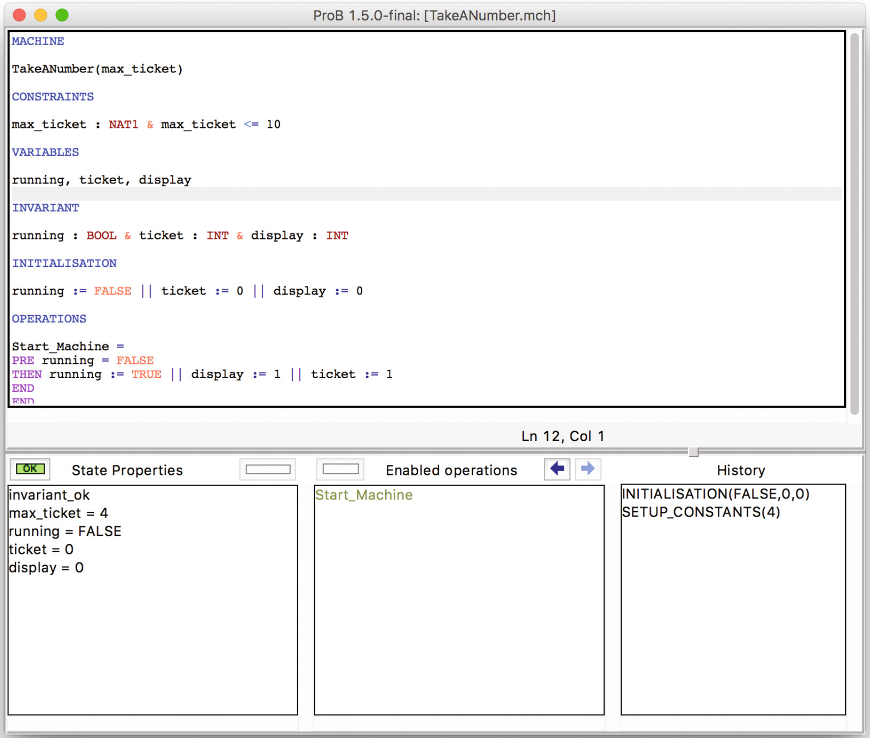
The Then steps are used to specify the state to test after the execution of the scenario. This diverges from the use of “then” in a typical BDD development process. Using a traditional BDD approach, “Then” would contain a number of assertions that would be made on the system-under-development after the “when” step occurred. As BHive is synthesizing a B Machine, we use a conjunction of the predicates specified using the method `context.state.test_that` to generate test cases via the ProB tool.

3.2 Synthesis of “Start Machine”

Invoking Behave extended with BHive synthesizes a B machine containing our variable definitions and our newly specified “Start Machine” operation. The output log is shown below:

```
11/01/2016 12:26:53 PM    before_all
11/01/2016 12:26:53 PM    registering b-types
11/01/2016 12:26:53 PM    Registered system type: BOOL
11/01/2016 12:26:53 PM    Registered system type: INT
11/01/2016 12:26:53 PM    Registered system type: NAT
11/01/2016 12:26:53 PM    Registered system type: NAT1
11/01/2016 12:26:53 PM    before_feature: Ticketing
11/01/2016 12:26:53 PM    registered machine: TakeANumber
11/01/2016 12:26:53 PM    before_scenario: Start_Machine
11/01/2016 12:26:53 PM    before_step: Machine is off
11/01/2016 12:26:53 PM    after_step: Machine is off
11/01/2016 12:26:53 PM    before_step: Machine is started
11/01/2016 12:26:53 PM    after_step: Machine is started
11/01/2016 12:26:53 PM    before_step: Machine is on
11/01/2016 12:26:53 PM    after_step: Machine is on
11/01/2016 12:26:53 PM    before_step: Display shows 1
11/01/2016 12:26:53 PM    after_step: Display shows 1
11/01/2016 12:26:53 PM    before_step: Ticket is 1
11/01/2016 12:26:53 PM    after_step: Ticket is 1
11/01/2016 12:26:53 PM    after_scenario: Start_Machine
11/01/2016 12:26:53 PM    after_feature: Ticketing
11/01/2016 12:26:53 PM    after_all
11/01/2016 12:26:53 PM    synthesizing
11/01/2016 12:26:53 PM    Test Predicate: running = TRUE
& display = 1 & ticket = 1
```

We can open and animate this specification with the ProB tool, depicted in Fig. 4.



```
MACHINE

TakeANumber(max_ticket)

CONSTRAINTS

max_ticket : NAT1 & max_ticket <= 10

VARIABLES

running, ticket, display

INVARIANT

running : BOOL & ticket : INT & display : INT

INITIALISATION

running := FALSE || ticket := 0 || display := 0

OPERATIONS

Start_Machine =
PRE running = FALSE
THEN running := TRUE || display := 1 || ticket := 1
END
END
```

```
invariant_ok
max_ticket = 4
running = FALSE
ticket = 0
display = 0
```

```
Start_Machine
```

```
INITIALISATION(FALSE,0,0)
SETUP_CONSTANTS(4)
```

Fig. 4. Screenshot of ProB tool

In the output log we see BHive registering B's primitive types (BOOL, INT, NAT, NAT1), and running all features and scenarios to build its internal model for state changes. Finally, this model is synthesized to B Method, and predicates to be used for ProB's test generation tool are emitted.

Completed Mocks. Shown below are the completed mocks using BHive's framework:

```
from behave import given, when, then

from bhive.integration import log_info, declare_variable

@given(u'Machine is off')
def step_impl(context):
    declare_variable(context, 'running', 'BOOL', 'FALSE')
    declare_variable(context, 'display', 'INT', '0')
    declare_variable(context, 'ticket', 'INT', '0')
    context.state.ensure_that('running','=','FALSE')

@when(u'Machine is started')
def step_impl(context):
    context.state.assign([('running', ':=', 'TRUE')])
    context.state.assign([('display', ':=', '1')])
    context.state.assign([('ticket', ':=', '1')])

@then(u'Machine is on')
def step_impl(context):
    context.state.test_that(context, 'running', '=',
'TRUE')

@then(u'Display shows 1')
def step_impl(context):
    context.state.test_that(context, 'display', '=', '1')

@then(u'Ticket is 1')
def step_impl(context):
    context.state.test_that(context, 'ticket', '=', '1')

@given(u'Machine is on')
def step_impl(context):
    context.state.ensure_that('running', '=', 'TRUE')

@when(u'Machine is stopped')
def step_impl(context):
    context.state.assign([('running', ':=', 'FALSE')])
    context.state.assign([('display', ':=', '0')])
```

```
@then(u'Machine is off')
def step_impl(context):
    context.state.test_that(context, 'running', '=',
'FALSE')

@then(u'Display shows 0')
def step_impl(context):
    context.state.test_that(context, 'display', '=', '0')

@when(u'Customer takes <ticket>')
def step_impl(context):
    context.state.assign([('ticket', ':=', 'ticket +
1')])

@then(u'Display should be less than or equal to the
ticket')
def step_impl(context):
    context.state.test_that(context, 'display', '<=',
'ticket')
    context.state.test_that_always(context, 'display',
'<=', 'ticket')

@given(u'Display is less than <ticket>')
def step_impl(context):
    context.state.ensure_that('display', '<', 'ticket')

@when(u'A customer is called')
def step_impl(context):
    context.state.assign([('display', ':=', 'display +
1')])

@when(u'Machine is reset')
def step_impl(context):
    context.state.assign([('display', ':=', '1')])
    context.state.assign([('ticket', ':=', '1')])
```

B Machine. Synthesis with BHive yields the following B Machine:[2]

[2] max_ticket is a parameter manually introduced in the synthesis process for TakeANumber to limit state explosion during model checking activities.

```
MACHINE
TakeANumber(max_ticket)2

CONSTRAINTS
max_ticket : NAT1 & max_ticket <= 10

VARIABLES
running, ticket, display

INVARIANT
running : BOOL & ticket : INT & display : INT & display
<= ticket

INITIALISATION
running := FALSE || ticket := 0 || display := 0

OPERATIONS

Customer_Takes_Ticket =
PRE running = TRUE
THEN ticket := ticket + 1
END;

Reset_Machine =
PRE running = TRUE
THEN display := 1 || ticket := 1
END;

Start_Machine =
PRE running = FALSE
THEN running := TRUE || display := 1 || ticket := 1
END;

Serve =
PRE running = TRUE & display < ticket
THEN display := display + 1
END;

Stop_Machine =
PRE running = TRUE
THEN running := FALSE || display := 0
END
END
```

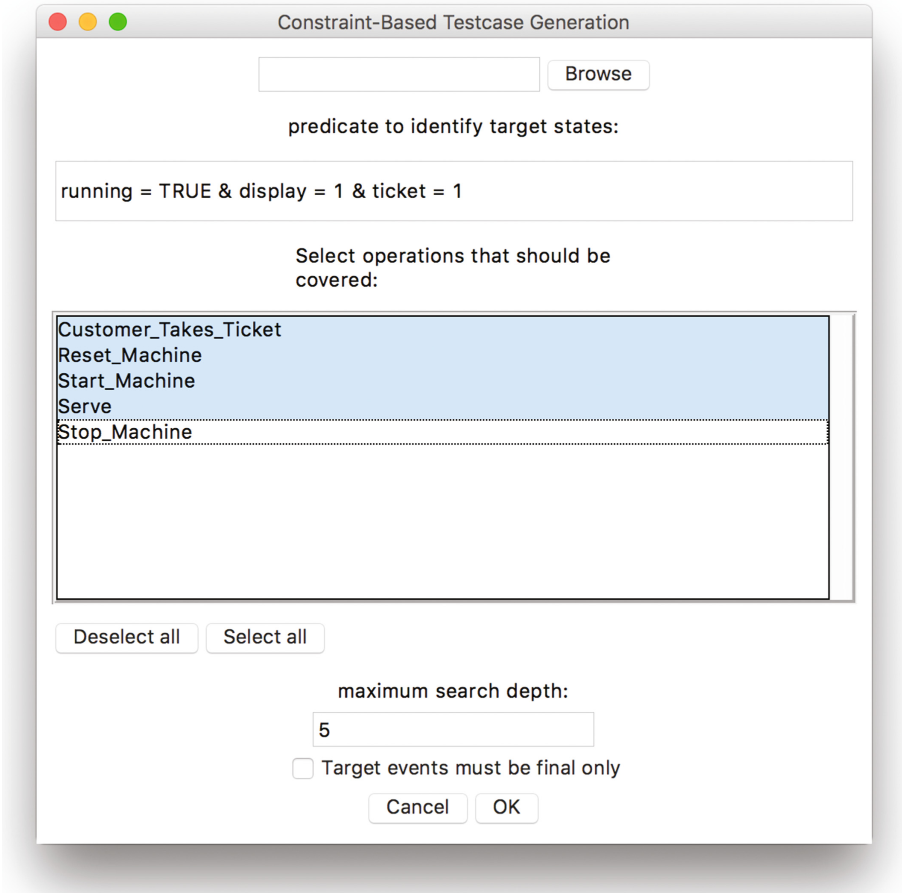
4 BHive's Approaches to Verification

BHive presently supports two approaches to verification, both involving ProB, which are described next.

4.1 ProB's Test Generation

By including a context.state.test_that() invocation in a Then step, BHive will synthesize the associated predicate to be fed into ProB's for test case generation (either manually using the ProB GUI or the command line via an XML file).

ProB's Test Case generation specifies a predicate and a set of operations. ProB will then generate a finite number of test cases with system traces that satisfy this predicate


Constraint-Based Testcase Generation

Browse

predicate to identify target states:

running = TRUE & display = 1 & ticket = 1

Select operations that should be covered:

Customer_Takes_Ticket
Reset_Machine
Start_Machine
Serve
Stop_Machine

Deselect all
Select all

maximum search depth:

5

Target events must be final only

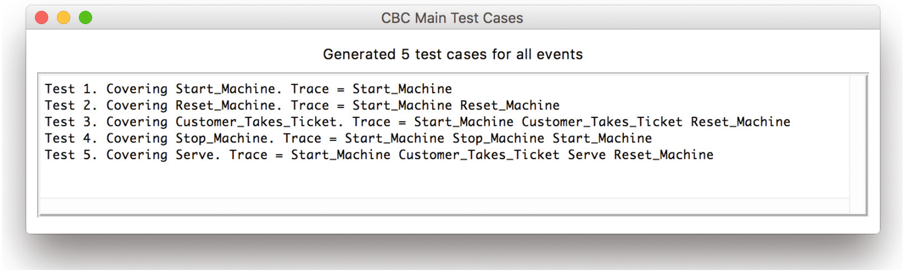
Cancel
OK

Fig. 5. Constraint-based test case generation using ProB

using a specific set of operations. For example, in “Start Machine” the Then steps produce the following predicate:

`running = TRUE & display = 1 & ticket = 1`

We can input this into ProB’s constraint-based-checking (CBC) test case generation tool as shown in Fig. 5, and select a set of operations to be covered. ProB will then generate a set of test cases where the provided predicate is satisfied, shown in Fig. 6.



Generated 5 test cases for all events

```
Test 1. Covering Start_Machine. Trace = Start_Machine
Test 2. Covering Reset_Machine. Trace = Start_Machine Reset_Machine
Test 3. Covering Customer_Takes_Ticket. Trace = Start_Machine Customer_Takes_Ticket Reset_Machine
Test 4. Covering Stop_Machine. Trace = Start_Machine Stop_Machine Start_Machine
Test 5. Covering Serve. Trace = Start_Machine Customer_Takes_Ticket Serve Reset_Machine
```

Fig. 6. Results of CBC test case generation with ProB

ProB supports both constraint-based-checking (CBC) and model-based-checking (MCM). CBC does not construct the entire state space, instead using ProB’s constraint solver to find relevant traces of operations. Both approaches have advantages and disadvantages, however, a detailed discussion is outside the scope of this work.

4.2 Handling of Invariants

At present, invariants are introduced into the synthesized B Machine in two ways:

- Typing: When a variable is declared, an invariant is added for that variable’s corresponding type.
- TEST_THAT_ALWAYS()

By including TEST_THAT_ALWAYS() additional invariants can be added to the specified machine. For instance, in the “TakeANumber” machine, we add to the invariant:

`display <= ticket`

ProB provides an easy interface to check the synthesized model for invariant violations, and can provide example traces to any such violation states.

5 BHive’s Development Flow

Figure 7 shows a block diagram of the major components of the BHive development flow. They are explained below:

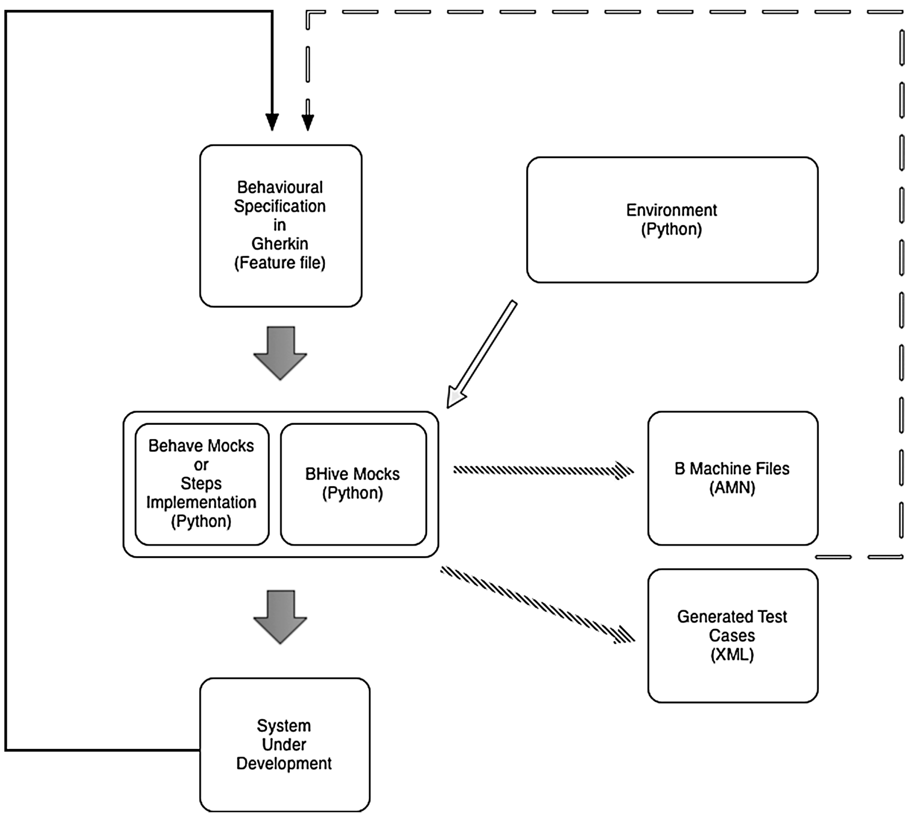

Behavioural Specification in Gherkin (Feature file)
Environment (Python)
Behave Mocks or Steps Implementation (Python)
BHive Mocks (Python)
B Machine Files (AMN)
Generated Test Cases (XML)
System Under Development

Fig. 7. BHive development flow

- A feature is specified behaviorally in a Gherkin feature file. From this file, the Behave tool can be invoked to generate stubs for all necessary steps.
- Initially, a developer would implement the steps using BHive mocks exclusively. The BHive mocks, plus typing information implemented in the environment file, allow a B Machine file to be created.
- The B Machine can be explored and animated, e.g., using ProB [11], with the latter's built-in model checker used to formally verify the invariant and display a trace that leads to a violation, which can be confirmed by invoking that path using Python code. Any other incorrect or underspecified behavior would inform revisions to the Gherkin feature file (shown by the dashed line), then verification can be repeated.
- Once the developer is happy with the feature they have explored using the BHive mocks, they can flesh out the system using Python's built-in mocks. Alternatively, they may choose to implement directly, or have even skipped the BHive mocking altogether in cases where a formal model would not offer additional value. BHive mocks can coexist alongside Python's built in mocks, or an actual implementation by disabling BHive when invoking behave.

- As the system under development takes shape, a clearer design emerges which feeds back into the Gherkin feature files. This transition is shown by the solid black line.
- Assertions contained in the "Then" steps for scenarios are synthesizes using ProB's [11] built-in test case generation. Based on the assertions specified, generated test cases will provide sequences of state changes that satisfy a given predicate. Manual testing can then be performed using these test cases. ProB will also report cases where the given predicate is not reachable. ProB uses a constraint based approach to test case generation, generating traces of operations that cover certain specified operations and satisfy a given predicate.

Note that this development flow does not explicitly call for producing source code by means of refining the B Machine. While this is possible, it requires the involvement of B-Method gurus, and it is more likely that agile practitioners would be satisfied with building on the mocks. The compromise at the heart of the BHive approach is that developers can enjoy some benefits of the injected formalism, compared to using BDD alone, but need not commit to the path of full formal development with all the costs that entails.

The system under development need not be a Python implementation, but rather any language that is capable of providing Python bindings. Cython is a compiler that provides two-way communication between C/C++ and Python, or SWIG creates general purpose bindings for a variety of target languages to be called from Behave step functions.

6 Application of BHive to System Development

BHive, like BDD, is best suited for control dominated systems and is not readily applicable to data-driven applications; such applications are difficult to express purely as externally observable behavior.

For example, a "take-a-number" ticketing system can easily be described to a developer creating such a system by outlining its functionality and providing examples of possible interactions, using a relatively small set of examples for typical use cases and others selected to illustrate edge case behavior. By contrast, a system intended for data processing and transformation such as a signal processing application would be highly cumbersome to specify using means other than the underlying mathematical equations.

For systems amenable to BDD development, synthesizing a specification from a set of Gherkin feature files provides an early preview of how the designed system handles state. The use of a model animator (such as ProB) to view and explore a model requires minimal training and offers a tangible way for stakeholders to "check their work" concerning the system they've described. By having a model that can be animated, a stakeholder can uncover cases of underspecification and improve the outlines passed to their scenario examples. The generation of test cases provides a "head start" for manual testing teams and could serve to highlight non-obvious execution paths a given state.

7 Related Work Integrating Formal and Agile Methods

"An Agile Formal Development Methodology" [12], published two years after the Agile Manifesto [2], proposed XFun, an extension to finite state machines built upon the Unified Process (UP) [13]. The authors acknowledge that existing formal methods are at a disadvantage in highly dynamic development approaches where requirement changes may invalidate previous models. XFun is an iterative approach that follows the phases of UP—Inception, Elaboration, Construction, Transition—through some iterations with the following activities:

1. Requirements gathering.
2. Construction of models from system requirements using XFun.
3. Application of XFun to animation and verification of constructed models as well as the generation of test cases.
4. A phase where the system is implemented.
5. A validation phase where the implementation is checked against the previously XFun-generated test cases.

Assuming successful application of UP and decomposition for iterations, the authors offer the ability of a formally verified system to evolve without the need for a single up-front modelling activity that is brittle with respect to late-breaking project changes. Despite being an iterative process, the work presented is comparatively sequential when compared with more recent approaches including TDD and BDD. The approach is similar to BHive in that animation and verification of constructed models is central, as well as the generation of test cases from those models. Aside from the choice of formalism and of development process (UP), the biggest difference between XFun and BHive is the potential disconnect with stakeholders when the project evolves from requirements gathering to model constructions. With BHive the models are constructed from the mediums used to capture requirements, Gherkin.

In 2004, Ostroff et al. presented "Agile Specification-Driven Development" [14]. The work identifies traditional development processes, characterized by a need for complete requirements up front, as "plan-driven development." Their approach is novel in that:

- It builds upon a plan-driven approach of "design by contract" (DbC) [15], where a system is constructed using a schema of preconditions, postconditions, and invariants which align to P, C, and Q (respectively) present in a Hoare Triple.
- It applies "Test-Driven Development" [16] (also known as "Test First" development) alongside the DbC specification.

The authors acknowledge that these two constructs had been regarded as "absolute extreme opposites with no combination possible or desirable," but show that DbC and TDD are complementary in the sense that they provide different capabilities for specification. The application of TDD is significant: TDD grew out of the Extreme Programming [17] movement and was applied across a variety of methods. Ostroff et al. highlight that a set of tests form a specification. Passing tests demonstrates conformance to the specification. Refactoring activities can be performed with confidence that a refinement of the system may be checked against the set of tests previously passed. TDD, which they describe as suitable for "collaborative specification," is tempered with additional assurances made possible by the rigor of DbC. The work presented by Ostroff is very encouraging for the BHive approach of bridging formalism and agile development process. DbC is semantically similar to B-Method with the inclusion of preconditions, postconditions and invariants. TDD is a close relative of BDD, and is frequently combined with BDD during development. BHive offers the advantage of a close integration between the formalism and the requirements medium (BDD's Gherkin). With Ostroff's work, the formalism of DbC and agility of TDD exist in two spheres, and the proficiency of the a practitioner familiar with both TDD and DbC determines the success of the integration. In contrast, with BHive's formalism is derived from the requirements medium at the outset, removing much of the "space" between the formal aspects and agile development approaches.

In 2006, Lopez-Nores et al. introduced "An Agile Approach to Support Incremental Development of Requirements Specification" [18] which describes an iterative approach to requirements refinement, not unlike model refinement methods used in B-Method and Event-B. Their approach is agile in that it "exploits the characteristic volatility of the early stages of development, to establish a frequent dialog with the stakeholders." With their approach, each iteration of analysis is refined through model checking, with amendments arising out of model checking activities proposed to stakeholders as a refinement to what was previously specified. The approach is agile in that, given a specification, additional requirements can be added to the model without the need for construction of a new model. Instead, through a series of refinements and abstractions, the model is adapted to include new requirements. The approach described is successful at being agile, due to its iterative nature but is more successful at bringing an agile perspective to formal development, rather than injecting formalism and its associated correctness into the world of agile development. Secondly, the requirements specification is subject to additional constraints, making it less appropriate to serve as a ubiquitous language between all stakeholders as is possible with Gherkin.

Rutledge et al. proposed Formal Specification-Driven Development (FSDD) [19] building upon TDD and BDD through the introduction of an additional artifact: "a formal design specification expressed in a behavioral specification language." The work identified the following shortcomings of BDD:

- A BDD specification is comprised solely of scenarios limiting the creators and maintainers in what they can express which they argue is contrary to Evan's notion of a Ubiquitous Language [5] and is prone to "knowledge transfer errors" as found without using a UL.
- When considering the difference between BDD and TDD, the authors assert: "Without the expansion into the analysis and test realms, BDD devolves into TDD with a specific vocabulary."

Rutledge et al. cite a study [20] showing no gains to code quality and statistically insignificant productivity boost through the application of TDD. The authors of that study explain this through the scope of the project used in the study and the inexperience of its participants. Rutledge et al. offer an additional explanation: TDD does not address knowledge transfer issues which contribute to overall quality of the system and acknowledge that any process designed to supplant or augment TDD and BDD must provide tool support that allows frequent and automated execution of the created tests/specifications. They outline their approach which is comprised of the following steps and artifacts:

- A designer creates a state-based formal specification from a set of requirements, aided by a “design specification analyzer.” From the formal specification artifact, the following artifacts are created:
  - A coder creates the software manually. He or she is able to begin with a set of stub functions generated automatically from the preceding formal specification.
  - The developer is aided by a set of unit tests automatically generated from the formal specification using an FSDD-specific unit testing framework.
  - QA analysts test the system using a manually created test plan that has been analyzed by an FSDD-supported test case analyzer.

Rutledge et al. [19] provide a sound methodology base on which many formal methods could be integrated into a more traditional plan-based development approach. Formalism in FSDD is introduced as part of an agile development process, in the form of assertions and predicates in the source files themselves. Though this offers additional assurance of correctness, this approach relies heavily on the developers and designers to “learn to read and write a new, more abstract language. Additionally, designers with an implementation background will have to adopt a new approach. They must learn to think in terms of ‘what’ an entity does, rather than ‘how’ an entity does it.” BHive is advantageous in that the formalism is derived from the ubiquitous language, rather than requiring retraining in a language that may not be appropriate for all stakeholders.

Lastly, a 2009 article “Formal Versus Agile: Survival of the Fittest” [21] surveys the state of integrating formal methods with agile methods. The authors acknowledge potential disconnects between agile and formal method communities, but identify where the two communities share common goals: testing, requirements specification, documentation, and parallelism. They identify two conditions in order for formalism to achieve wide-scale agile “buy-in”: speed and availability of such methods, and the need for “flexible tools.”

BHive is novel when compared to surveyed work in that it injects formalism into an existing agile development technique to form a very close coupling of two ostensibly incompatible approaches to system development (the agile process, and the plan driven world of formal specification and verification). Through allowing the development team to determine how much formalism to introduce into their Gherkin specifications, the team is able to inject formalism incrementally and use discoveries found by examining generated models to inform future development. What’s more, the BHive process is compatible with existing BDD practices, so our methodology may be added when it offers value, or removed without requiring rework.

8 Conclusion and Future Work

Here we have described our ongoing work in connecting an unexploited formalism underlying Gherkin's "Given-When-Then" syntax with Abrial's B-Method. In doing this, we have bridged the world of behavior specification suitable for close collaboration with domain experts and non-technical stakeholders, with the world of high quality, formally verified systems.

Future efforts will focus on the following:

- Improving the internal models used by the BHive mocking library, particularly the ability to introduce additional invariants.
- Enrichment of the typing system, including building types from set comprehensions. Presently only enumerations are supported.
- Migration from "classic B" to Event-B, which will allow the creation of theorems to prove specific properties on models.
- B offers very flexible options for machine reuse and structuring. The modular nature of B's AMN provides the ability for incremental development. BDD's feature structure and contained scenarios can be synthesized into a self-contained machine and validated in isolation. Future enhancements of BHive will seek to leverage B's decomposition mechanisms.
- Improved and automatic selection of parameters for automatic test case generation. Presently a "one size fits" all approach is used. This can be improved by embedding some of ProB's test case generation semantics into BHive itself.

References

1. Carter, J., Gardner, W.: BHive: towards behaviour-driven development supported by B-Method. In: Proceedings of 4th IEEE International Workshop of Formal Methods Integration (FMi) at IEEE 17th International Conference on Information Reuse and Integration (IRI), Pittsburgh, PA, pp. 249–256. IEEE, July 2016
2. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: Manifesto for Agile Software Development (2001), http://www.agilemanifesto.org/
3. Forsberg, K., Mooz, H.: The relationship of system engineering to the project cycle. In: Proceedings of the National Council for Systems Engineering First Annual Conference, pp. 57–61 (1991)
4. North, D.: Behavior modification: the evolution of behaviour-driven development. Better Softw. Mag., March 2006. "Introducing BDD"
5. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Longman, Boston (2003)
6. Gherkin wiki (2016), https://github.com/cucumber/cucumber/wiki/Gherkin
7. Rice, B., Jones, R., Enge, J.: Welcome to behave! (2014), http://pythonhosted.org/behave/
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)

9. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
10. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (2010)
11. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From animation to data validation: the ProB constraint solver 10 years on. In: Boulanger, J.-L. (ed.) Formal Methods Applied to Complex Systems: Implementation of the B Method, chap. 14, pp. 427–446, Wiley ISTE, Hoboken (2014)
12. Eleftherakis, G., Cowling, A.J.: An agile formal development methodology. In: Proceedings of 1st South-East European Workshop on Formal Methods (SEEFM 2003), pp. 36–47 (2003)
13. Scott, K.: The Unified Process Explained. Addison-Wesley Longman, Boston (2002)
14. Ostroff, J.S., Makalsky, D., Paige, R.F.: Agile specification-driven development. In: Eckstein, J., Baumeister, H. (eds.) Proceedings of Extreme Programming and Agile Processes in Software Engineering (XP 2004), vol. 3092. Lecture Notes in Computer Science, pp. 104–112. Springer (2004)
15. Meyer, B.: Applying "design by contract". Computer **25**, 40–51 (1992)
16. Beck, K.: Test-Driven Development: By Example. Addison-Wesley Longman, Boston (2002)
17. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley Professional, Boston (2004)
18. Lopez-Nores, M., Pazos-Arias, J.J., Garcia-Duque, J., Barragans-Martinez, B.: An agile approach to support incremental development of requirements specifications. In: Proceedings of Australian Software Engineering Conference (ASWEC 2006), Washington, DC, pp. 9–18. IEEE Computer Society (2006)
19. Rutledge, R., Duggins, S., Lo, D., Tsui, F.: Formal specification-driven development. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP), p. 1, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) (2014)
20. Erdogmus, H., Morisio, M., Torchiano, M.: On the effectiveness of the test-first approach to programming. IEEE Trans. Softw. Eng. **31**, 226–237 (2005)
21. Black, S., Boca, P.P., Bowen, J.P., Gorman, J., Hinchey, M.: Formal versus agile: survival of the fittest. Computer **42**, 37–45 (2009)

A Pre-processing Tool for Z2SAL to Broaden Support for Model Checking Z Specifications

Maria Ulfah Siregar[1,2(✉)]

[1] Department of Computer Science, The University of Sheffield, Regent Court, 211 Portobello, Sheffield S1 4DP, UK
acp12mus@sheffield.ac.uk

[2] Informatics Department, UIN Sunan Kalijaga, Yogyakarta, Indonesia

Abstract. One of the deficiencies of Z tools is that there is limited support for model checking Z specifications. Building a model checker directly for a Z specification will take considerable amount of effort and time due to the abstraction of the language. Translating a Z specification input into a specification in a language that an existing model checker tool accepts is an alternative method. Researchers at the University of Sheffield implemented a translation tool, Z2SAL, that takes a Z specification and translates it into the input for Symbolic Analysis Laboratory (SAL), a framework for combining different tools for abstraction, program analysis, theorem proving and model checking. This paper discusses support for model checking Z specifications, in which the capability of Z2SAL is extended. This support includes a translation of a generic constant and a schema calculus definition. Instead of translating these aspects of the Z language into the SAL language as Z2SAL does, a Z specification containing these two notations will be pre-processed, in which a generic constant definition is redefined to an equivalent axiomatic definition and a schema calculus definition is expanded to a new schema definition. As a result of a successful redefinition or expansion, a redefined or expanded Z specification is generated, otherwise the original Z specification is returned. Results show that the large number of our examples can be run successfully by our system. The redefined or expanded Z specification can be translated later by Z2SAL and the generated SAL file can be model checked or simulated by the SAL tool. Results also show that Z2SAL can translate outputs of our system to some extent. The majority of generated SAL files can be run by the SAL tool.

Keywords: Z · Specification · Generic constant · Schema calculus · Z2SAL · SAL

1 Introduction

As a formal language, the use of Z in academia and industry has increased considerably. This is because Z has been used successfully to address a large variety of problems and the international standard was also designed for this language. The use of Z can make a specification more formal and free from ambiguity. In addition, Z allows a specification to be analysed mechanically [1]. Designing a specification of a system enables a user to verify the system at an early stage of development. Early verification could avoid high cost of system implementation and test phases, if the specification was designed correctly [2–4]. Therefore, a specification is crucial for a system, especially if the system relates to the safety of property and/or life.

However, there is a lack of tools for this language, especially in model checking Z specifications. Although the Community Z Tools (CZT) project is developing a set of open source tools for Z, progress of this development is slow [5]. There are many causes of the shortage of Z tools. These are mostly related to the Z language and semantics, such as an inherent expressiveness and a difficulty in deciding effectively any theorem about Z specifications [1,5]. Another cause is the richness of this language, which can also be the issue of verifying Z [1]. Furthermore, only a few of these tools can be used in validating intended meanings of such Z specifications [6].

The lack of supporting tools for the Z language and the above issues has led researchers suggest alternative methods, which is a more rapid approach, to address this problem: reuse and adapt existing tools. Researchers at the University of Sheffield implemented the Z2SAL translator [5] which generates a SAL specification from a Z specification input. The generated SAL file can be model checked later by the SAL model checker. A brief introduction to Z2SAL and SAL is given in Sect. 2.

In our study, several experiments using Z2SAL and SAL are performed. Our finding is Z2SAL supports many Z tags, but not all. Furthermore, sometimes several generated SAL files cannot be verified or simulated by the SAL tool.

Therefore, this paper intends to address problems as stated below:

1. What are crucial features of Z should be implemented to enhance the ability of Z2SAL and why?
2. How to implement such features that are supported by Z2SAL and SAL?

These questions will be explored in the following sub-sections. Both the below sub-sections did not exist in [12].

1.1 Motivation

Based on our experiences using Z2SAL, two aspects of the Z notation were chosen to study. Both aspects will be discussed in this section.

The first aspect is the Z generic construct. Z2SAL cannot translate specifications that consist of generic constructs. As a result, error files were generated instead. Our finding is that Z2SAL cannot recognize a generic constant which is one type of the Z generic constructs. Although it has been declared in the generic constant definition, Z2SAL reported that the generic constant is a new identifier.

Z2SAL has not encountered any generic construct on Z specifications beforehand, so this part of Z has not been implemented yet. Therefore, our assumption is that the current version of Z2SAL does not support translation of either a generic constant or a generic schema. Although, Z2SAL's researchers might implement them.

Our study in the SAL literature concludes that a generic form cannot be found in the SAL language. Thus, another assumption is that Z2SAL does not support generic constructs in order to be consistent with the SAL language.

Specified using generic parameters, a generic constant is commonly used in formulating mathematical tool-kit operators [7], in which the operators do not depend on the particular type of elements in their construction [8]. Another usage of a generic constant is to specify a general notion which is used frequently in a system.

In case there is no generic constant, several equivalent functions should be formulated because each function is dedicated to one set of types of parameters; it is redundant work. Thus, a generic constant is beneficial to a Z specification.

The second aspect is the Z schema calculus. Z2SAL supports a translation of several schema calculus such as a schema inclusion, the Δ operator, and the Ξ operator. However, they must be specified either vertically or horizontally in a schema. If a new schema is specified as being constructed from earlier schemas, Z2SAL does not support this schema construction. Thus, it is argued that Z2SAL does not support schema calculus.

The constructed schema is used commonly to define a more complex, modular and larger specification of a system. Therefore, schemas are reused to create new schemas. These schemas are combined by using schema operators. Different schema operators which are used define different new schemas.

Therefore, focus was set on generic constants and schema calculus as crucial features of the Z notation in our work. They were studied to extend Z2SAL so it can translate both of them. These findings were used to define our objective as discussed below.

1.2 Objective and Contribution

Our objective is to implement a tool. This tool will redefine a generic constant definition to an equivalent axiomatic definition based on usages of this generic constant.

Another objective is to implement a tool to create a new schema by expanding other schemas. These schemas are connected by schema operators.

Both these tools are implemented in a system which is called support for model checking Z specifications. This System is our contribution to broaden the applicability of model checking Z specifications. JFlex [9], BYACC/J [10], and Java language [11] were used to implement our system. During our experiments with this system, there is another contribution of a SAL translation of user defined functions and constants. It will be discussed later.

The paper is organized as follows. Section 2 describes briefly an introduction to Z2SAL and SAL. Section 3 contains our support for model checking

Z specifications. This section also discusses how to implement this support which is supported by both Z2SAL and SAL. It is divided into two sub-sections. Each sub-section has been extended from its previous version in [12]. Section 3.1 outlines our support for generic constants. There are several new sub-sections in this section, such as Generic Abbreviation Definitions, Lambda Expressions, Summaries of Experiments on the Redefinition System, Size of Z Specifications, and Manual Modification in SAL files. However, majority of the contents of the first three ones and the last one have been discussed in [12]. The new sub-section, Size of Z Specifications, discusses how our redefinition system scales to a variety of sizes of Z specifications. Section 3.2 explains another support which is schema calculus. There is a new sub-section in this section, Summaries of Experiments with the Expansion System. Several contents of this sub-section were gathered from earlier sub-section. There are several new experiments have been performed in the expansion system as compared to the previous version in [12]. These experiments were summarised in new tables accompanied the new sub-section. Section 4 concludes this paper and summarises future work. This last section has also been extended from its previous version in [12].

2 A Brief Introduction to Z2SAL and SAL

Several tools in Z have been developed based on the quick approach, such as *ProZ* [6] which is a translator of Z into the existing Alloy Analyser tool, *ProB* [13]; data refinement verification [14] which uses Alloy SAT-solver based on a counter-example finder; and Z2SAL [5] which is a translator of a Z language specification into a SAL language specification [15].

Smith and Wildman at the University of Queensland, Australia, described how to translate a Z language specification into a SAL input language specification [16]. This basic idea was implemented in a tool set [17] and the current Z2SAL extends it in a different direction, to tackle optimization issues [5].

In providing a translator of Z into an input language of existing tools, SAL was chosen since it has an *equivalent representation* of many aspects of Z [17]. Moreover, 'many different tools exist, which use the SAL input language such as simulator, model checker either symbolic or bounded, deadlock checker, etc' [5], which are offered freely by Stanford Research Institute (SRI) International under academic licences.

A generated SAL file consists of a SAL module and/or several SAL contexts. This module describes a transition system of Z states [17]. The simple SAL module has a general format as follows:

```
State : MODULE =
  BEGIN
     INPUT ...
     LOCAL ...
     OUTPUT ...
     INITIALIZATION [ ... ]
     TRANSITION [
        ...
     ]
  END;
```

The SAL context is a place to declare types, constants, modules and modules properties [15]. Z2SAL formulates several Z mathematical tool-kits, which are necessary for a generated SAL specification, in separate but integrated SAL context files.

Translating a Z language specification into a SAL input language specification requires several adjustments due to a number of differences of both languages [5]. These adjustments are discussed briefly as given below:

First, is *bounding the infinite*. Z supports *fully abstract* (non-grounded, non-constructive) specification styles, whereas SAL is a *concrete and grounded language*. For example, Z supports the built-in numerical types "$\mathbb{Z}$", "$\mathbb{N}$" and "$\mathbb{N}_1$", whose ranges are infinite. On the other hand, SAL has similar unbounded types `INTEGER`, `NATURAL` and `NZNATURAL`, which can be used only as base types of finite sub-ranges in a SAL specification. Z also supports given types, which have semantics of an un-interpreted set, such as `[TAPE, NAME]`. Therefore, the translations provided by Z2SAL should specify a finite number for sizes of these sets.

The *mismatched formal paradigms* are the second difference. Z and SAL have very different styles of specifications and descriptions. The Z specification, which consists of state schemas and operational schemas, is built-up increasingly. It views locally and functionally such that every operational schema operates on its input and output variables, or on variables of state schemas. On the other hand, the SAL specification is created as a 'monolithic finite state automaton' (FSA) such that all inputs, outputs and local variables are compiled into aggregate states [5]. Moreover, all operations act upon guard transitions from one state configuration to another state configuration [5]. Thus, this mismatch can be approached by re-ordering all information in the Z specification. A further mismatch is that Z specifications often use partial functions [5]. On the other hand, as SAL is based on *Binary Decision Diagrams* (BDDs), SAL always requires a representation of function as a total function. Thus, a work-around is necessary in order to present a partial function in Z specifications as a total function in SAL. Furthermore, a set cannot be treated as a monolithic FSA of SAL, but as a 'poly-lithic collection of judgements' over its elements instead [5]. Thus, several operations in sets are necessary to be expressed differently, such as the cardinality of a set, which is not supported by SAL.

The last difference is an issue of *non-computable specifications* [5]. A Z specification naturally supports non-constructive styles of a specification. These styles should be expressed in computable styles of a specification in SAL. Both styles essentially are different indeed. Normally, a SAL specification consists of a set of update assignments to primed variables, which indicates posterior variable states. On the other hand, a direction of a constructive approach is not necessary in a Z specification. Z2SAL asserts posterior existences of variables and restricts their values on preconditions. This requires a search for suitable precondition values.

More information relating to Z2SAL is provided in [18]. It also includes a downloadable version of this translation tool.

SAL is a framework of several different tools such as abstraction, program analysis, theorem proving and model checking, which is used to change concepts and implementations of model checkers and theorem provers. These concepts and implementations at first were based on verification, but they were extended to include calculation of properties or symbolic analysis such as abstraction, slicing and composition [15,19].

The SAL language can be used as a specification language, a target language for several translators, or a common source of several analysis tools. It is originated of a collaboration of two researchers, David Dill from Stanford University and Thomas Henzinger from the University of California at Berkeley. These collaborations devolved SAL further and incorporated Verimag. SAL is now developed at SRI and its current version is 3.3. The SAL language syntax can be found in [15].

The next section describes our support for model checking Z specifications.

3 Support for Model Checking Z Specifications

As mentioned earlier in Sect. 1.2, there are two main types of our support for model checking Z specifications. The first is a generic constant, which will be described in the following sub-section.

3.1 Support for Generic Constants

Our first support is to aid Z2SAL to translate generic constants. The following sub-sections describe briefly an introduction to a generic constant and our system, also discuss results of several examples.

Introduction. A generic constant is used to introduce a new constant which uses generic parameters [7]. By using a generic parameter, different types of a parameter can be specified. They are specified by using different literals such as `X`, `Y`, `Z` and others. A generic constant has a global scope in a Z specification, whereas a generic parameter has a local scope in the particular generic constant definition.

An example of a generic constant definition is formulated as follows:

$$\begin{array}{|l}
[X] \\
\hline
\mathit{monoSequence} : \mathbb{P}(\mathrm{seq} X) \\
\hline
\mathit{monoSequence} = \{ s : \mathrm{seq} X \mid \#(\mathrm{ran}\ s) \leq 1 \} \\
\hline
\end{array}$$

The above definition has `monoSequence` as the generic constant, which is a constant (see a discussion below). The output type is a set of a sequence of `X`. There is one specified generic parameter, `X`. This generic constant definition defines a set of a sequence of `s`, which just has at the most one element.

A Generic Constants Redefinition System. Our approach to support Z2SAL in translating generic constant definitions is to implement a tool. This tool will redefine a generic constant definition to an equivalent axiomatic definition based on usages of this generic constant (see Sect. 1.2). This approach is based on similar behaviour between a generic constant and an axiomatic definition. In other words, they both declare a global variable inside a Z specification. This redefinition is called an actualization process, in which a generic typed parameter will be actualised to its actual typed parameter.

Plagge and Leuschel in [13] also proposed the same method as our method for translating a generic definition defined in a Z specification. As discussed in their paper, generic constant definitions had not been added to Z specification examples.

Our system specifies different types of generic constants. These types can be identified based on the generic constant declarations, as given below:

- a function; the outermost operator is an infix generic function. A complete set of these functions is "⇸", "→", "⤔", "↣", "⤀", "↠" and "⤖". These functions are collected in one token, `INGEN`. As a function, it will have at least one input parameter and one output parameter. This type can be generic.
- a relation; a declaration uses the "↔" tag in its outermost operator. This tag has the `REL` string as its token. As a relation, there is no output parameter type. In other words, the output is the relation itself; a pair of types.
- a constant; a constant means it does not require any input. Thus, a declaration of this generic constant only gives us generic output parameters. This declaration denotes none of the above tags in its outermost declaration.

The above three types of generic constants are parts of a variable declaration of the Z grammar in the Z language. This grammar, which refers to [8], was specified in our parser as follows:

```
expr1:   expr1.word REL decor expr1.word
     |   expr1.word INGEN decor expr1.word
     |   expr2.chain
     |   expr2
     ;
```

The first production rule indicates a relation, whereas the second one is a function. The third production rule contains `CROSS` obtained from `expr2.chain`. Thus, this production rule can either be a function or a relation depending on which of those first two production rules is fired previously. The last one is a constant; both function and relation production rules are not matched.

Inevitably, a constant actualization is not always straightforward, especially a constant implicit type. In this case, a solution is to infer the actual type of the generic constant.

Our redefinition system is intended as a pre-processing tool which can aid Z2SAL. A Z specification input, which consists of generic constant definitions and usages, will be pre-processed by this tool in order to redefine its generic constant definitions.

This tool was implemented in Java. It has a simple GUI to interact with users and has also two preliminary processes: the scanner and the parser generation. These two generators were implemented by using the JFlex scanner generator [9] and the BYACC/J parser generator [10] respectively.

The current version of our system implemented several Z tokens which refer to [8,20] and several production rules of the Z grammar which refer to [8]. Our system also experienced of simple variable types of generic constants.

The next sub-section discusses an example of the redefinition process. This Z specification was taken from [21], **the swap function**.

An Example of the Redefinition Process. This specification has one given type, `NAME`. There are two generic constant definitions for the swap process defined in this specification. These functions, each of which has two parameters, swap the orders of its parameters. Thus, after a swap, an element in the second position will be shifted such that this element is in the first position and vice versa.

The first definition, as shown below, has two different generic parameters: `X` and `Y`. These different parameters mean that both of them have different types. The generic constant is `swap2` which is shown in the following example:

$$
\begin{array}{|l}
[X, Y] \\
\hline
swap2 : X \times Y \rightarrow Y \times X \\
\hline
\forall\, x : X;\ y : Y \bullet swap2(x, y) = (y, x) \\
\hline
\end{array}
$$

The second definition has one generic parameter, `X`. This single parameter means that the swap process will occur on objects of the same type. The generic constant is `swap1` which is shown in the following example:

$$
\begin{array}{|l}
[X] \\
\hline
swap1 : X \times X \rightarrow X \times X \\
\hline
\forall\, x, y : X \bullet swap1(x, y) = (y, x) \\
\hline
\end{array}
$$

A state schema, `State`, has only one state variable, `name`, which is an instance of the specified given type. There is no predicate specified in this schema.

The initialization schema, `Init`, refers to the post state of the state schema. This schema does not declare its own variable and predicate. It means that this schema only contains predicates which are inherited from its reference to the state schema. In this case, the reference is the post state of the state schema.

There is one operational schema specified in this specification, `Swap`, which calls these generic constants. This schema does not change a state of the system indicated by a reference to Ξ `State`. This schema is specified as follows:

$$
\begin{array}{|l}
\textit{Swap} \\
\hline
a? : \mathit{NAME};\ a!, b! : \mathit{NAME};\ c? : \mathbb{N};\ c! : \mathbb{N};\ \Xi \mathit{State} \\
\hline
(b!, a!) = \mathit{swap1}[\mathit{NAME}, \mathit{NAME}](\mathit{name}, a?) \\
(c!, a!) = \mathit{swap2}(\mathit{name}, c?) \\
\hline
\end{array}
$$

As can be seen in the above schema, each generic constant has one usage. The first usage uses explicit parameter types in addition to parameters required by the function. Our system generates two axiomatic definitions for these usages as shown below:

$$
\begin{array}{|l}
\mathit{swap1} : \mathit{NAME} \times \mathit{NAME} \rightarrow \mathit{NAME} \times \mathit{NAME} \\
\hline
\forall x, y : \mathit{NAME} \bullet \mathit{swap1}(x, y) = (y, x)
\end{array}
$$

$$
\begin{array}{|l}
\mathit{swap2} : \mathit{NAME} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathit{NAME} \\
\hline
\forall x : \mathit{NAME};\ y : \mathbb{N} \bullet \mathit{swap2}(x, y) = (y, x)
\end{array}
$$

Consider that the explicit type has been deleted from the first usage since Z2SAL does not support this type of parameter. Thus, the first usage should be modified by our system as follows:

$$(b!, a!) = \mathit{swap1}(\mathit{name}, a?)$$

This modification was conducted on this usage to let Z2SAL translates this specification successfully.

Result and Discussion. The generated specification of the above example can be translated by Z2SAL. A SAL file, generated by Z2SAL, can also be verified by the SAL model checker. However, it failed to be simulated by the SAL model checker. This simulator generated an unsupported error of a failure to convert function application.

Furthermore, if a theorem was added to the generated SAL file, this SAL file cannot be verified either by the SAL model checker. Thus, it is an issue of the redefinition system.

The current Z2SAL translates the Z functions, relations and constants, and puts them in the base module. Z2SAL defines `State` as the default name for this module. The simple structure of this module can be seen in Sect. 2. This translator also puts variable declarations in a definition clause. The definition clause is part of the base module or in other words it is inside the base module.

As a result, an error was sometimes experienced during our experiments with user-defined functions. This error related to an incompatible type in the equality operator or a failure to convert function application produced by the SAL model checker or simulator, as given earlier.

A user defined function, relation and constant are always declared outside a SAL module [15]. They are put in a context clause, specifically in a constant declaration, instead. The module language in SAL describes transition system modules [15]. However, it cannot be used to declare new types or constants or asserting properties of the module [15]. All of these can be easily declared by specifying them in the SAL context language.

A translation method of user defined functions adapted by the SAL language is different to the one that Z2SAL adapts. Based on this finding, the same method as SAL's method was proposed by us to Z2SAL researchers during our study. This proposal can be considered as our contribution in model checking Z specification as mentioned in Sect. 1.2.

This method can be applied to a user defined function and constant, but it is not applicable to a user defined relation since a relation does not have a type for its output parameter. It is based on a signature of this SAL function specified in [15].

The signature of which is named as a constant declaration has the following syntax rule [15]:

$$\mathit{ConstantDeclaration} := \mathit{Identifier}[(\mathit{VarDecls})] : \mathit{Type}[= \mathit{Expression}]$$

This constant declaration, as mentioned above, is part of the SAL context language. The SAL context language syntax is given as follows [15]:

$$\begin{array}{lll}
\mathit{Context} & ::= & \mathit{Identifier}[\{\mathit{Parameters}\}] : \mathit{CONTEXT} = \mathit{ContextBody} \\
\mathit{Parameters} & ::= & [\mathit{TypeDecls}]; \{\mathit{VarDecls}\}^{*,} \\
\mathit{TypeDecls} & ::= & \{\mathit{Identifier}\}^{+,} : \mathit{TYPE} \\
\mathit{ContextBody} & ::= & \mathit{BEGINDeclarationsEND} \\
\mathit{Declarations} & ::= & \mathit{ConstantDeclaration} \\
 & \mid & \mathit{TypeDeclaration} \\
 & \mid & \mathit{AssertionDeclaration} \\
 & \mid & \mathit{ContextDeclaration} \\
 & \mid & \mathit{ModuleDeclaration} \\
\mathit{ConstantDeclaration} & ::= & \mathit{Identifier}[(\mathit{VarDecls})] : \mathit{Type}[= \mathit{Expression}] \\
\mathit{TypeDeclaration} & ::= & \mathit{Identifier} : \mathit{TYPE}[= \mathit{TypeDef}] \\
\mathit{AssertionDeclaration} & ::= & \mathit{Identifier} : \mathit{AssertionForm} = \mathit{AssertionExpression} \\
\mathit{AssertionForm} & ::= & \mathit{OBLIGATION} \mid \mathit{CLAIM} \mid \mathit{LEMMA} \mid \mathit{THEOREM} \\
\mathit{ContextDeclaration} & ::= & \mathit{Identifier} : \mathit{CONTEXT} = \mathit{Identifier}\{\mathit{ActualParameters}\} \\
\mathit{ActualParameters} & ::= & \{\mathit{Type}\}^{*,}; \{\mathit{Expression}\}^{*,}
\end{array}$$

Other non-terminals or rules can be found in the same reference as given above.

Thus, the generated SAL file was modified to adapt a constant declaration formulated by SAL. Both the above function definitions were formulated manually on the generated SAL file. They are shown below:

```
swap1(q__1 : NAME, q__2 : NAME): B__NAME__X__B__NAME = (q__2, q__1);

swap2(q__3 : NAME, q__4 : NAT): B__NAT__X__B__NAME = (q__4, q__3);
```

Original declarations generated by Z2SAL for these functions were deleted.

A few theorems were added to this specification as shown below:

```
th1: theorem State |- G(FORALL (i: NAME, j: NAT): swap2(i,j) = (j,i));

th2: theorem State |- G(FORALL (i,j: NAME): i = j =>
                      swap1(i,j) = swap1(j,i));

th3: theorem State |- G(FORALL (i,j: NAME): swap1(i,j) = swap1(j,i));
```

The first two theorems are valid; the swap system can satisfy both properties. The last theorem is invalid since the swap function will not give us the same result for different parameters.

There is another issue relating to an abbreviation definition and a lambda expression which was found during our experiments with the redefinition system. Both these issues will be discussed in the next sub-sections.

Generic Abbreviation Definitions. Z2SAL supports an abbreviation definition, but not the generic one. Declaring a global constant by using an abbreviation definition is common in writing Z specifications. Thus, a generic abbreviation definition was taken also into our consideration.

In the case of generic abbreviations, it is not enough just to work with an actualization of a generic type. The other issue here is a set comprehension definition. A generic abbreviation definition is usually defined by using a set comprehension definition. However, Z2SAL does not support an abbreviation definition consisting of a set comprehension.

For example, consider a generic abbreviation definition as below [7]:

$$monoSequence[X] == \{s : \mathrm{seq} X \mid \#(\mathrm{ran}\ s) \leq 1\}$$

A generic abbreviation definition can be rewritten to a generic constant definition. Both these definitions declare global constants in the related Z specification, in this case the type of the generic constant is a constant.

The expression in the right hand side of the "==" uses a set comprehension definition, which denotes that `monoSequence` is a set of a sequence of `X`. The body of this generic definition is obtained from the expression after the "==" tag.

Thus, a generic abbreviation definition is first rewritten to a generic constant definition. This rewriting is performed manually and automatically in order to prove that it is correct. This equivalent definition was given in Sect. 3.1. Afterwards, this generic constant definition is redefined to an axiomatic definition.

Lambda Expressions. Another kind of generic forms is the "λ" expression, which is used to define a function without specifying a name [7]. Z2SAL does not support this expression which is common in generic constant definitions or in other definitions in a Z specification generally. Our approach is to rewrite a lambda expression automatically and manually to an equivalent expression without any lambda expression. Then, it is redefined to an axiomatic definition.

For example, a generic constant definition as formulated below consists of the lambda expression [7]:

$$\begin{array}{|l}
[X] \\
\hline
commonSubseq : ((\mathrm{seq} X) \times (\mathrm{seq} X)) \rightarrow \mathbb{P}(\mathrm{seq} X) \\
\hline
commonSubseq = (\lambda\, s, t : \mathrm{seq} X \bullet allSubseqs \cap allSubseqt) \\
\hline
\end{array}$$

The lambda expression in the above definition can be rewritten to an equivalent definition as follows:

$$commonSubseq = \{s, t : \mathrm{seq} X \bullet ((s, t), allSubseq(s) \cap allSubseq(t))\}$$

or another equivalent one as given below:

$$\forall\, s, t : \mathrm{seq} X \bullet commonSubseq(s, t) = allSubseqs \cap allSubseqt$$

A lambda expression definition, (λ `S • E`), represents a function and has arguments which are taken from `S`. An output of this expression is the value of `E` [8]. As given by the first equivalent definition above, the lambda expression is equivalent to a set comprehension, $\{S \bullet (T, E)\}$, in which `T` is a characteristic tuple of `S`. In a set comprehension, a characteristic tuple is obtained from its declaration. Thus, `(s,t)` is the characteristic tuple of the above set comprehension.

During our experiments, Z2SAL was unable to translate a set comprehension definition with many parameters of the same type. According to the SAL grammar rules, only one parameter can be declared in one definition of a set comprehension. The SAL syntax [15] for a set expression is given as follows:

$$\begin{array}{l}
SetExpression := SetListExpression \mid SetPredExpression \\
SetListExpression := \{\{Expression\}^{+}_{,}\} \\
SetPredExpression := \{Identifier : Type = Expression\}
\end{array}$$

Thus, our approach is to rewrite the first equivalent lambda expression to the second equivalent one.

Several results collected from our experiments are summarized and discussed in the next sub-section.

Summaries of Experiments on the Redefinition System. A number of experiments on several Z specifications are presented on Table 1. These experiments run on a laptop with a 1.30 GHz Genuine Intel(R) CPU U7300 and 2.00 GB RAM.

The second column of Table 1 indicates that a manual modification was made to the SAL file. The SAL file was generated by Z2SAL from the Z specification

Table 1. Several experiments with the redefinition system

Z Specification	Details	Verification time in secs	
(*.tex)		#Theorem = 0	#Theorem > **0**
`bbook`	Modified SAL function		0.842
`bbook_map`	Modified SAL function	0.016	0.25
`bbook_uni`	Modified SAL function and other parts of SAL file	0.031	0.406
`bbook_map_uni`	Modified SAL function and other parts of SAL file		0.359
`fDomRan`	Modified SAL function	0.015	
`fEmpty`	OK		0.093
`fEmptyImpl`	OK		0.109
`fFirst`	Modified SAL function	0.015	0.187
`fHead`	Modified SAL function	0.031	
`fHeadFunc`	Modified SAL function and cannot be simulated: The set of initial states is empty	0.031	
`fMaxComSubSeq`	Modified other parts of SAL file and cannot be simula ted: An out of memory error	0.047	
`fMaxComSubSeq_1`	Modified other parts of SAL file and cannot be simula ted: An out of memory error	0.032	
`fMaxComSubSeq_orig`	Modified other parts of SAL file and cannot be simula ted: An out of memory error	0.032	
`fMonoSeq`	OK. Long simulation	0.047	
`fMonoSeq_1`	OK. Long simulation	0.031	
`fSwap`	Modified SAL function	0.016	0.141
`fUniqSeq`	Ok. Cannot be simulated: An out of memory error	0.062	
`fUniq1Seq`	Ok. Cannot be simulated: An out of memory error	0.031	
`fUniq2Seq`	Ok. Cannot be simulated: An out of memory error	0.015	
`tn`	Modified other parts of SAL file and cannot be simula ted: An out of memory error	0.03	
`tnImpl`	Modified other parts of SAL file and cannot be simula ted: An out of memory error	0.0	
`fFileStorage`	Canot be translated by Z2SAL	N/A	
`fSet`	Modified SAL function and other parts of SAL file	0.0	

produced by our redefinition system. This modification is required so that the SAL file can be verified by the SAL model checker or simulated by the SAL simulator. It involved rewriting a user defined function and placing this function in which SAL put its function. Examples of this rewriting were given earlier in Sect. 3.1. The modification also involved rewriting other parts of a SAL file. Such a modification implies that there is a bug in the translation of associated Z specification by Z2SAL. It can also be a mismatch between the Z language and the SAL language. This manual modification will be discussed in the later sub-section.

The third column shows verification times of each SAL file. A SAL file which has one verification time means that this file has only be verified for one case of the number of theorems. A SAL file which has two verification times means that the SAL file at first can be verified by the SAL model checker. However, later it cannot be verified if at least one theorem was added to this SAL file. Such a SAL file usually cannot be simulated either by the SAL simulator even there is no theorem.

Based on our experiments as shown in Table 1, majority of SAL files generated by Z2SAL from the output of our system, can be verified by the SAL model checker. It is proved by existences of a verification time in each row of the table.

Inevitably, there is one output produced by our system which cannot be translated by Z2SAL. The output is generated from the **fFileStorage.tex** input file. It is because this Z specification contains a function which its range is also a function. Z2SAL does not support such a type. A quick solution is to rewrite such a function. However, another error relating to the ". ." tag was experienced. It was concluded that it is a bug in a Z2SAL translation of a range of numbers.

Another finding is that a few SAL files, which can be verified by the SAL model checker, cannot be simulated by the SAL simulator due to an out of memory error as can be seen in Table 1. These SAL files usually have sequences or a set inside other sets. Currently, this error has not been solved.

Relating to out of memory errors, the following sub-section discusses this issue. The discussion will be accompanied by a table.

Size of Z Specifications. Z specifications used for our experiments have different sizes measured in kilobytes. These sizes are summarized in Table 2. Sizes of the redefined specifications are recorded also on this table.

"`input`" on this table means a Z specification input file for our system. On the other hand, "`output`" means a Z specification output file generated by our system after performing a redefinition process, "N/A"s in `SAL specifications` means an associated Z specification cannot be translated by Z2SAL.

Referring to this table, almost all of our experiments have the same sizes of Z specifications before and after redefinition processes. It means that there were not many usages specified in these specifications. It can also mean that the generic constant definitions are not quite complex definitions.

The size of a SAL specification is roughly twice to four times of its Z specification. Sizes of SAL specifications shown in this table are original sizes producing

Table 2. Sizes of Z specifications

Z Specifications	Sizes in KB		
(.tex)	Input	Output	SAL specifications
`bbook`	2	2	6
`bbook_map`	1	1	4
`bbook_uni`	1	1	4
`bbook_map_uni`	1	2	5
`fDomRan`	2	2	6
`fEmpty`	1	1	2
`fEmptyImpl`	1	1	2
`fFirst`	1	1	3
`fHead`	1	1	3
`fHeadFunc`	1	1	3
`fMaxComSubSeq`	2	2	4
`fMaxComSubSeq_1`	2	2	4
`fMaxComSubSeq_orig`	2	2	4
`fMonoSeq`	1	1	3
`fMonoSeq_1`	1	1	3
`fSwap`	1	1	2
`fUniqSeq`	1	2	5
`fUniq1Seq`	1	2	5
`fUniq2Seq`	1	2	5
`tn`	3	3	6
`tnImpl`	3	3	6
`fFileStorage`	2	2	N/A
`fSet`	2	2	5

by Z2SAL. As discussed above, several of these SAL specifications have been modified as required in order to be executed by the SAL tool successfully. Thus, their sizes can be different from the original ones.

There are only four experiments which their sizes of Z specification outputs were increased twice of their inputs. These specifications are `bbook_map_uni`, `fUniqSeq`, `fUniq1Seq` and `fUniq2Seq`.

As shown in Table 1, several of our Z specifications cannot be simulated by the SAL simulator because of out of memory errors. However, these errors cannot be blamed for the increasing sizes of specifications. It is because there are other specifications which their sizes were not increased, but they were involved on the same errors as above. Sizes of these specifications are greater than 1. However, it can coincide which is not influenced entirely only by sizes of specifications.

An additional factor is the complexity of a declaration of a generic constant. The out of memory errors were involved on specifications which have either sequences, or sets of sets.

Thus, a Z specification, which does not have a sequence, a set of other set, or a range of numbers, can be executed successfully by the SAL tool. It argues also that a size of a generic constant definition and a number of usages relate to a generation of that error.

In a conclusion, our approach to redefine generic constants definitions scales to larger specifications. However, as the outcomes of our system will be translated by Z2SAL and executed by the SAL tool later, the large specification generated by our system is possible to be a problem with both tools.

As mentioned earlier, a separate discussion in manual modification in SAL files will be offered. The following sub-section discusses our approach to this manual modification.

Manual Modification in SAL Files. Although all generated SAL files in our experiments with this system can be verified by the SAL model checker, a few of them at first failed. The modified version of these SAL files also failed to be verified by the SAL model checker. These files are **output_bbook_uni** as the SAL file generated from the output of **bbook_uni.tex**, **output_bbook_map_uni** as the SAL file generated from the output of **bbook_map_uni.tex** and **output_fSet** as the SAL file generated from the output of **fSet.tex**.

This failure related to incompatible types in the equality operator. The SAL model checker identified that the type of `birthday` is not compatible with the type of the first argument of a function `uniSet` in the first and second SAL files. The `uniSet` function which is a generic constant definition was specified as follows:

$$
\begin{array}{|l}
[X] \\
\hline
\mathit{uniSet} : (\mathbb{P} X) \times (\mathbb{P} X) \rightarrow (\mathbb{P} X) \\
\hline
\forall S, T : (\mathbb{P} X) \bullet \mathit{uniSet}(S, T) = \{x : X \mid x \in S \lor x \in T\} \\
\hline
\end{array}
$$

This function combines two sets of elements which have the same types. As can be seen, this function requires two parameter inputs. Both of them have the same types as the output type.

An example of usage of the above generic constant is specified as follows:

$$\mathit{birthday}' = \mathit{uniSet}(\mathit{birthday}, \{\mathit{name}? \mapsto \mathit{date}?\})$$

As can be seen from the above generic constant definition, the type for the first parameter is a set of `X`. This type is an expected type. On the other hand, `birthday` is the first parameter passed to `uniSet`. The type of `birthday` will be the actual type for this parameter. The declaration of the function `birthday` is as follows:

$$birthday : NAME \pfun DATE$$

`birthday` is a state variable, which is a partial function from `NAME` to `DATE`.

Our system generated the `uniSet` axiomatic definition as follows:

$$\begin{array}{|l} uniSet : (\mathbb{P}(NAME \times DATE)) \times (\mathbb{P}(NAME \times DATE)) \rightarrow (\mathbb{P}(NAME \times DATE)) \\ \hline \forall\, S, T : (\mathbb{P}(NAME \times DATE)) \bullet uniSet(S, T) = \{x : (NAME \times DATE) \mid x \in S \lor x \in T\} \end{array}$$

As can been from the above definition, the type of `birthday` has been modified to its equivalent type. It is done so to ease the unification of the expected type, `X`, and the actual type, `NAME` $\pfun$ `DATE`.

A function type can be rewritten to a relation type [8]. Several constraints should also be added to maintain that it was a function. Furthermore, a relation is equivalent to a set of a pair of types.

$$X \leftrightarrow Y \equiv \mathbb{P}(X \times Y)$$

Thus, SAL failed to recognize that `birthday` had an equivalent type to the first argument of the `uniSet` user-defined function. This error indicated that there was incompatible type between the output of `uniSet`, `Set__C__B__NAME__X__B__DATE__I`, and the right hand side of the equality operator, `[NAME__X__DATE -> bool]`. Afterwards, a sequence of modifications was performed to the associated SAL file lines.

The last error produced by the SAL model checker is as follows:

```
Error: [Context: output_bbook_uni_mod, line(62), column(29)]:
Type mismatch in the function application.
Expected type:
[set{output_bbook_uni_mod!NAME__X__DATE}!Set,
set{output_bbook_uni_mod!NAME__X__DATE}!Set]
Actual type:
[output_bbook_uni_mod!Set__C__NAME__X__B__DATE__I,
set{output_bbook_uni_mod!NAME__X__DATE}!Set]
```

The related SAL lines are as follows:

```
61 NOT set {NAME;} ! contains?(known, name?) AND
62 birthday' = uniSet((birthday, set {NAME__X__DATE;} !
63 singleton((name?, date?)))) AND
64 invariant__'
```

In line 62, the type of `uniSet` after modification is a pair of `set {NAME__X__DATE;} ! Set` and `set {NAME__X__DATE;} ! Set`. This type was not compatible with the actual type passed to `uniSet`, which was a pair of `Set__C__NAME__X__B__DATE__I` and `set {NAME__X__DATE;} ! Set`. The type `Set__C__NAME__X__B__DATE__I` is an alias for `[NAME -> B__DATE]`, specified by Z2SAL.

Although a function is special type of a relation and a relation is a set of a pair of types in the Z language, SAL did not conclude that both types of the first argument of `uniSet` are the same. Thus, this incompatible type was solved manually. This is because our tool has not been able to perform this modification automatically.

Our last modification defined the same alias for `birthday`, but this time the alias represents a relation, not a function any more. It is shown as follows:

```
Set__C__NAME__X__B__DATE__I : TYPE = set {NAME_X_DATE;} ! Set;.
```

This change affects the usage of `birthday`. It cannot any longer be used as a function.

```
function {NAME, B_DATE; DATE_BB} ! partial?(birthday) AND
```

As a result, the above line was deleted from the old SAL file.

```
known = relation {NAME, DATE;} ! domain(birthday) AND
```

Another result is the above line, which is a relation, replaced a line, which is a function, as follows:

```
known = function {NAME, B_DATE; DATE_BB} !  domain(birthday) AND
```

As well as a line as follows, which presents a usage of a function:

```
date' = birthday(name?) AND
```

was replaced by a line below, which presents a usage of a relation:

```
set {NAME_X_DATE;} ! contains? (birthday, (name?, date')) AND
```

Finally, the modified SAL file can be verified by the SAL model checker and simulated by the SAL simulator.

The same function was also a source of the error in the third SAL file, but this time its first actual parameter is `used`. A usage of this function in the associated Z specification is as follows:

$$used' = uniSet(used, n)$$

The `used` state variable is a set of "$\mathbb{N}_1$" and the `n` variable is an instance of "$\mathbb{N}_1$".

An axiomatic definition generated for the above usage is as follows:

$$uniSet : (\mathbb{P}\mathbb{N}_1) \times (\mathbb{P}\mathbb{N}_1) \rightarrow (\mathbb{P}\mathbb{N}_1)$$
$$\forall S, T : \mathbb{P}\mathbb{N}_1 \bullet uniSet(S, T) = \{x : \mathbb{N}_1 \mid x \in S \vee x \in T\}$$

After a similar modification as performed in both SAL files above, the modified SAL file can be verified and simulated by the SAL tool.

Another aspect of the Z notation in our study is the schema calculus. This aspect was taken as the second type of our support for model checking Z specifications.

3.2 Support for Schema Calculus

This sub-section discusses the addition of support for Z schema calculus to our tool. The sub-section begins with an introduction to schema calculus. It is followed by a brief introduction to our support for this Z notation and our experiments on this system.

Introduction. Z2SAL supports a translation of several schema calculus such as a schema inclusion, the Δ and the Ξ operator, but they must be specified either vertically or horizontally in a schema. However, if a new schema is constructed from earlier schemas, Z2SAL does not support this schema construction. Thus, it argues that Z2SAL does not support schema calculus definitions.

The constructed schema is specified by using "$\widehat{=}$". It is the same as the supported schema calculus. However, the constructed schema does not use "[" and "]" to surround its declaration of variables and predicates.

The constructed schema is used commonly to define a more complex, modular and larger specification of a system. Schemas that have been specified can be reused to specify a new schema. It is because every schema has its distinctive operation in a specification, called 'schema separation' [2].

A Schema Calculus Expansion System. Our approach is to construct a new schema by expanding other schemas, in which they are connected by schema operators. This system was included in the support tool for model checking Z specifications, the same as the redefinition system.

Since every schema operator has its own definition, a schema operator affects how the expansion is performed. The expansion means that all unique variables of involved schemas are listed in the new schema. It also means that predicates from the involved schemas are added. These predicates are combined using specified schema operators.

There is a prerequisite for operating two schemas; the same or common variables should have the same type. Furthermore, in a case of the negation operator, normalisation is also required.

Normalisation is to define explicitly the constraint given by the declaration part of the related schema. This constraint is specified in the predicate part. Normalisation should be performed just before the negation. This process is applied also to other schema operators for the sake of easiness. Several normalisation rules were specified in our system as follows:

- Every "$\mathbb{N}$" or "$\mathbb{N}_1$" in a declaration part is rewritten to a type of "$\mathbb{Z}$".
- Every "seq" or "seq_1" is changed to $\mathbb{P}(\mathbb{Z} \times \mathit{newVal})$, newVal is a type which comes after "seq" or "seq_1". The previous rule is applied also to *newVal*.
- Every function is changed to a pair of its left hand side type and its right hand side one. Both the above rules are also applied to the type in the left and in the right.

In general, after each schema is expanded, variables and predicates will be collapsed to a reference of a state schema. This collapse benefits the new schema to get a more compact schema and to avoid re-declarations of state variables.

Our system can expand several schema operators such as conjunction "$\wedge$", disjunction "$\vee$", negation "$\neg$", implication "$\Rightarrow$", bi-implication "$\Leftrightarrow$", hiding "$\backslash$", renaming "/", composition "$\fatsemi$", universal quantifier "$\forall$" and existential quantifier "$\exists$". Our system can also perform a simple simplification over a predicate part.

The next sub-section describes an example from our experiments with this system.

An Experiment with the Schema Calculus Expansion System. An example, **expandingschema_3.tex**, will be presented in this sub-section. This example was taken from [2], but has been modified in some places for our experiments.

This example represents a library system specification. It has a state and an initialization schema as follows:

Library

$$\begin{array}{l}
\mathit{stock} : \mathit{COPY} \pfun \mathit{BOOK};\ \mathit{issued} : \mathit{COPY} \leftrightarrow \mathit{READER} \\
\mathit{shelved} : \mathbb{F}\,\mathit{COPY};\ \mathit{readers} : \mathbb{F}\,\mathit{READER} \\
\hline
\forall\, x : \mathit{COPY};\ y1, y2 : \mathit{READER} \bullet \\
(x \mapsto y1) \in \mathit{issued} \wedge (x \mapsto y2) \in \mathit{issued} \Rightarrow y1 = y2 \\
\mathit{shelved} \cup \mathrm{dom}\,\mathit{issued} = \mathrm{dom}\,\mathit{stock} \\
\mathit{shelved} \cap \mathrm{dom}\,\mathit{issued} = \varnothing \\
\mathrm{ran}\ \mathit{issued} \subseteq \mathit{readers} \\
\forall\, r : \mathit{readers} \bullet \#(\mathit{issued} \rhd \{r\}) \leq \mathit{maxloans}
\end{array}$$

InitLibrary

$$\begin{array}{l}
\mathit{Library}' \\
\hline
\mathit{shelved}' = \varnothing \\
\mathit{readers}' = \varnothing
\end{array}$$

As can be seen from the above state schema, this library system has four state variables:

- `stock` is a partial function from `COPY` to `BOOK`. It gives us information about what copies a book has.
- `issued` is a relation between a copy of a book and a reader. It gives us information about which copy of a book each reader has.

- `shelved` is finite set of `COPY`.
- `readers` is a finite set of `READERS`.

This system has also three given types: `COPY, BOOK, READER`.

There is one schema calculus definition specified in this specification, which uses the Z schema composition operator, "$\fatsemi$". It is shown as follows:

$$Donate \mathrel{\widehat=} EnterNewCopy \fatsemi RegisterReader.$$

This operator will combine the second schema with the first schema, in which the result of the first schema is an input for operating the second schema.

The schema composition consists of the number of operations taken from other schema operators. Renaming is the first operation to take place. Its processes begin with renaming the same state variables so that the primed ones in the first schema and non-primed ones in the second schema have the same name of variables. Afterwards, these renamed schemas are combined using a conjunction operator. The next process is to hide the common renamed variables in a declaration part of the new schema. It is followed by adding an existential quantification which binds these hidden variables in a predicate part of the new schema.

The new schema, `Donate`, was constructed by our system as given below:

Donate

$\Delta Library;\ b? : BOOK;\ r? : READER;\ rep! : Report$

$\exists\, c : COPY \mid c \notin \operatorname{dom} stock \bullet (stock' = stock \oplus \{c \mapsto b?\} \wedge$
$shelved' = shelved \cup \{c\}) \wedge r? \notin readers \Rightarrow$
$(readers' = readers \cup \{r?\} \wedge rep! = Ok) \wedge$
$r? \in readers \Rightarrow (readers' = readers \wedge rep! = ReaderAlreadyRegistered)$
$\wedge issued' = issued$

A theorem as given below was added to the generated SAL:

```
th1: theorem State |- G(shelved = set{COPY;}!empty);
```

It says that `shelved` is always empty, which is invalid. It is because `c` of type `COPY` can be added to `shelved` by performing `EnterNewCopy` or `Donate`. Indeed, the SAL model checker reported a counter-example on the verification of this SAL file.

This specification requires a simplification which is applied to the final output, otherwise there will be re-declared state variables. Our system could perform a simple simplification to collapse all state variables and predicates to a reference of the state schema.

As mentioned previously, the first process of a schema composition is renaming which is to rename several state variables to the common names. In this system, the common name is specified to be the same as the name of the state variable, but 0 will be added at the end of this variable. This simplification is achieved by substituting all renamed common variables for their appropriate values obtained from related predicates.

The above example can be translated by Z2SAL. It can also be verified and simulated by the SAL tool.

The following sub-section summarizes results obtained from our experiments with this system. A discussion in these results is also given in this sub-section.

Summaries of Experiments with the Expansion System. This sub-section discusses several findings found during our experiments with the expansion system. Each finding is discussed in a separate paragraph.

Re-declaration of State Variables. Re-declaration of state variables is an issue of implementations of renaming and hiding operations. Since a simplification is hard to apply on both operations, these operations cannot be further implemented at the moment.

The current Z2SAL assumes that the first schema definition in a Z specification is a state schema and the second one is an initialization schema. Z2SAL defines also one base module in each SAL specification and accepts only one state schema in each Z specification input, though both SAL and Z allow many modules and state schemas respectively in one specification.

A SAL module specifies a transition system of a finite-state automaton. A Z schema represents a state of a system and a collection of these schemas models behaviour of the system. A state schema is a combination of state variables and predicates of a system.

A restriction on the number of state schemas in a Z specification is also an issue of performing a negation in a schema expansion. Variables and negated predicates in the constructed schema cannot be collapsed into a state schema inclusion. It is because of a problem of re-declared variables. The only way to solve this problem is to define at least two state schemas. The first state schema just defines state variables, whereas the second one defines an inclusion to the first schema and state predicates. However, Z2SAL does not support many state schemas either as discussed earlier.

This restriction affects also how a renaming and a hiding are applied to. Both schema operators cannot be applied to the initialization schema and operational schemas due to the above same problem and instead to the state schema. Furthermore, Z2SAL also enforces us to define the same name for both the constructed schema and the state schema. Thus, the application of these two operators will modify the whole specification.

The Order of Schema Operators. Another issue in schema expansion is the order or the binding of schema operators, especially when brackets are not added in a definition of schema calculus. Fortunately, operators bindings and associativities can be defined by using built-in options of the BYACC/J parser generator [10]: `left`, `right` and `nonassoc`, which mean left, right and no grouping respectively. Afterwards, several actions can be added in associated grammars to define information about these orders. The order of operators tells us the precedence among them, which is getting higher position, the lower the precedence. Several of these orders are given in the [8].

Size of Z Specifications. One issue that is important to consider is by having many schema calculus definitions, both a Z specification and a SAL specification are also getting big in sizes. Another important issue is that a size of a SAL specification is roughly twice to four times of its Z specification (see discussion below).

Fortunately, our approach to expand schema calculus definitions scales to larger specifications. However, as the outcomes of our system will be translated by Z2SAL and executed by the SAL tool later, the large specification resulted by our system is possible to be a problem with both tools.

The following describes briefly our experiments with this system. Several tables are presented which summarize these experiments.

Tables 3, 4, and 5 show us results from several examples of our experiments. These examples were obtained from several Z books and they will be discussed below.

Specifications which were used for Experiment 1 to Experiment 8, and Experiment 71 were taken from [2]. It is a library system which has been discussed in Sect. 3.2.

On the other hand, Experiment 9, Experiment 24 to Experiment 33, Experiment 54 to Experiment 55, and Experiment 58 to Experiment 63 were taken from [22]. This is a simple car park system. The state schema and the initialization schema are as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CarsPark}} \\ \hline
count : \mathbb{N};\ maximum : \mathbb{N} \\ \hline
count \leq maximum \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitCarsPark}} \\ \hline
CarsPark \\ \hline
count = 0 \\
maximum = 3 \\ \hline
\end{array}$$

Experiment 10 to Experiment 14, Experiment 23, Experiment 34 to Experiment 44, Experiment 56 to Experiment 57, and Experiment 64 to Experiment 70 were taken from [4]. This system regards with bookings for performances on a concert hall. The state schema and the initialization schema for these experiments are as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{BoxOffice}} \\ \hline
seating : \mathbb{P}\, Seat \\
sold : Seat \twoheadrightarrow Customer \\ \hline
\mathrm{dom}\, sold \subseteq seating \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitBoxOffice}} \\ \hline
BoxOffice' \\ \hline
sold' = \varnothing \\
seating' = initial\_allocation \\ \hline
\end{array}$$

Experiment 15 to Experiment 22 were taken from [7].

Table 3. Several experiments with the expansion system

No	Z Specification	Details	Verification time in secs	
	(.tex)		Non-simplified	Simplified
1.	`expandingschema_1`	"$\vee$"	0.063	0.031
2.	`expandingschema_2`	"$\wedge$"	0.062	
3.	`expandingschema_3`	"$\fatsemi$"	0.03	
			0.733	
4.	`expandingschema_4`	"$\wedge$"	0.016	
		"$\vee$, $\vee$"		
		"$\vee$"		
			2.044	
5.	`expandingschema_5`	"$\wedge$, $\neg$, $\wedge$"	0.031	
			1.654	
6.	`expandingschema_6`	"$\wedge$, [,]"	0.031	
			0.686	
7.	`expandingschema_7`	"$\neg$, $\wedge$, [,]"	N/A	
8.	`expandingschema_8`	"$\wedge$, [,]"	N/A	
		"$\neg$, $\wedge$, [,]"		
		"$\vee$"		
9.	`expandingsch2_4`	"$\neg$"	N/A	
10.	`expandingsch3_1`	"$\Rightarrow$"	0.015	
11.	`expandingsch3_2`	"$\wedge$, $\Rightarrow$"	0.032	
12.	`expandingsch3_4`	"$\Rightarrow$, $\wedge$"	0.016	
13.	`expandingsch4_1`	"$\Leftrightarrow$"	0.015	
14.	`expandingsch4_2`	"$\wedge$, $\Leftrightarrow$"	0.031	
15.	`expandingsch5_1`	"[, /,]"	N/A	
16.	`expandingsch5_2`	"[, /, /,]"	N/A	
17.	`expandingsch6_1`	"\"	N/A	
18.	`expandingsch6_2`	"\"	N/A	
19.	`expandingsch7_1`	"$\fatsemi$"	0.031	
20.	`expandingsch8_1`	"$\forall$"	N/A	
21.	`expandingsch8_2`	"$\forall$"	N/A	
22.	`expandingsch8_3`	"$\forall$, $\wedge$"	N/A	
23.	`expandingsch8_6`	"$\exists$"	N/A	
24.	`expandingsch1_1`	"$\wedge$"	0.0	
25.	`expandingsch1_2`	"$\wedge$"	0.016	
26.	`expandingsch1_3`	"$\wedge$, $\wedge$, $\wedge$"	0.0	
27.	`expandingsch1_4`	"$\wedge$, $\wedge$"	0.015	
28.	`expandingsch1_5`	"$\vee$, $\wedge$"	0.015	
29.	`expandingsch1_6`	"$\vee$, $\wedge$"	0.0	
30.	`expandingsch1_7`	"$\wedge$, $\vee$"	0.015	
31.	`expandingsch1_8`	"$\wedge$, $\vee$"	0.0	

Table 4. Several experiments with the expansion system (continued)

No	Z Specification	Details	Verification time in secs	
	(.tex)		Non-simplified	Simplified
32.	expandingsch1_9	"$\wedge$"	0.0	
33.	expandingsch1_10	"$\vee$, $\vee$, $\vee$"	0.0	
34.	expandingsch1_11	"$\wedge$, $\vee$, $\wedge$"	0.016	
35.	expandingsch1_12	"$\wedge$, $\vee$, $\wedge$"	0.015	
36.	expandingsch1_13	"$\wedge$, $\vee$, $\wedge$"	0.016	
37.	expandingsch1_14	"$\wedge$, $\vee$, $\wedge$"	0.016	
38.	expandingsch1_15	"$\wedge$"	0.016	
39.	expandingsch1_16	"$\wedge$"	0.016	
40.	expandingsch1_17	"$\wedge$"	0.016	
41.	expandingsch1_18	"$\wedge$"	0.031	
42.	expandingsch1_19	"$\wedge$, $\vee$, $\wedge$"	0.032	
43.	expandingsch1_20	"$\wedge$, $\vee$, $\wedge$"	N/A	
44.	expandingsch1_21	"$\wedge$, $\vee$, $\wedge$"	0.03	
45.	expandingsch1_22	"$\wedge$, $\vee$"	0.031	
46.	expandingsch1_23	"$\wedge$, $\vee$"	0.032	
47.	expandingsch1_24	"$\wedge$, $\vee$"	0.031	
48.	expandingsch1_25	"$\wedge$, $\vee$"	0.016	
49.	expandingsch1_26	"$\wedge$"	0.015	
50.	expandingsch1_27	"$\wedge$"	0.031	
51.	expandingsch1_28	"$\vee$, $\vee$, $\vee$"	0.031	
52.	expandingsch1_29	"$\vee$, $\vee$, $\vee$"	0.031	
53.	expandingsch1_30	"$\vee$, $\vee$, $\vee$"	0.047	
54.	expandingsch1_31	"$\vee$, $\wedge$, $\vee$"	0.0	
55.	expandingsch1_32	"$\vee$, $\vee$, $\wedge$"	0.015	
56.	expandingsch2_1	"$\neg$"	N/A	
57.	expandingsch2_2	"$\neg$, $\wedge$"	0.032	
58.	expandingsch2_3	"$\neg$"	N/A	
59.	expandingsch2_5	"$\neg$, $\wedge$"	N/A	
60.	expandingsch2_6	"$\neg$, $\wedge$"	N/A	
61.	expandingsch2_7	"$\wedge$, $\neg$"	0.0	
62.	expandingsch2_8	"$\wedge$, $\neg$"	0.0	
63.	expandingsch2_9	"$\neg$, $\wedge$, $\neg$"	N/A	
64.	expandingsch3_3	"$\wedge$, $\Rightarrow$"	0.016	
65.	expandingsch3_5	"$\Rightarrow$, $\wedge$"	0.015	
66.	expandingsch3_6	"$\Rightarrow$, $\wedge$"	0.031	
67.	expandingsch3_7	"$\Rightarrow$, $\vee$, $\Rightarrow$"	N/A	
68.	expandingsch3_8	"$\wedge$, $\Rightarrow$, $\wedge$"	0.015	
69.	expandingsch3_9	"$\wedge$, $\Rightarrow$, $\wedge$, $\Rightarrow$, $\wedge$"	0.015	

Table 5. Several experiments with the expansion system (continued)

No	Z Specification	Details	Verification time in secs	
	(.tex)		Non-simplified	Simplified
70.	`expandingsch8_5`	"$\forall$"	N/A	
71.	`expandingschema_9`	"$\wedge$, $\neg$, $\wedge$"	N/A	
		"$\wedge$, [,]"		
		"$\neg$, $\wedge$, [,]"		
		"$\bigvee$"		
		"$\wedge$, $\vee$"		

Calculator
$store : MEMORY \rightarrow \mathbb{Z}$
$display : \mathbb{Z}$
$arg2 : \mathbb{Z}$

Init
$Calculator$

$\forall\, m : MEMORY \bullet store(m) = 0$
$display = 0$
$arg2 = 0$

Above are the state and initialization schemas of this specification. This specification is a system of a four function calculator.

Experiment 45 to Experiment 53 were taken from [23], but have been modified in several places to be able to be translated by Z2SAL. One of these modifications is to have one state schema. In the original specification, there are references to several different schemas. These references indicate the referenced schemas are state schemas. The modification is necessary to be translated by Z2SAL. A state and initialization schemas are given as follows:

Flexi
$Standard\_Hours, Flexitime\_Hours : Time \rightarrow Period$
$worked : Ident \twoheadrightarrow Period;\ in : Ident \twoheadrightarrow Time$

$\mathrm{dom}\, in \subseteq \mathrm{dom}\, worked$

InitFlexi
$Flexi$

$in = \varnothing$
$worked = \varnothing$

As can be seen from Tables 3, 4, and 5, a simplification has only been performed on an output of `expandingschema_1` specification. It is indicated by two verification times in associated columns. Outputs of `expandingschema_3`, `expandingschema_4`, `expandingschema_5`, and `expandingschema_6` have two

Table 6. Sizes of Z specifications

Z Specifications	Sizes in KB		
(.tex)	Input	Output	SAL specifications
Expandingschema_1	2	2	7
Expandingschema_2	2	2	6
Expandingschema_3	2	2	6
Expandingschema_4	2	3	10
Expandingschema_5	2	3	9
Expandingschema_6	2	2	4
Expandingschema_7	2	3	N/A
Expandingschema_8	3	5	N/A
Expandingsch2_4	1	N/A	N/A
Expandingsch3_1	1	1	3
Expandingsch3_2	2	2	4
Expandingsch3_4	2	2	4
Expandingsch4_1	2	2	4
Expandingsch4_2	2	2	5
Expandingsch5_1	1	1	N/A
Expandingsch5_2	1	1	N/A
Expandingsch6_1	1	1	N/A
Expandingsch6_2	2	2	N/A
Expandingsch7_1	1	1	2
Expandingsch8_1	2	2	N/A
Expandingsch8_2	2	2	N/A
Expandingsch8_3	2	2	N/A
Expandingsch8_6	1	2	N/A
Expandingsch1_1	1	1	3
Expandingsch1_2	1	1	3
Expandingsch1_3	1	1	3
Expandingsch1_4	1	1	3
Expandingsch1_5	1	1	3
Expandingsch1_6	1	1	3
Expandingsch1_7	1	1	3
Expandingsch1_8	1	1	3
Expandingsch1_9	1	1	3
Expandingsch1_10	1	1	3
Expandingsch1_11	1	2	3
Expandingsch1_12	1	2	4
Expandingsch1_13	1	2	4
Expandingsch1_14	1	2	3
Expandingsch1_15	1	1	3
Expandingsch1_16	1	1	3
Expandingsch1_17	1	1	3
Expandingsch1_18	1	1	3
Expandingsch1_19	2	2	4
Expandingsch1_20	2	N/A	N/A
Expandingsch1_21	2	2	4

Table 7. Sizes of Z specifications (continued)

Z Specification	Sizes in KB		
(.tex)	Input	Output	SAL specifications
`Expandingsch1_22`	2	2	4
`Expandingsch1_23`	2	2	4
`Expandingsch1_24`	2	2	4
`Expandingsch1_25`	2	2	5
`Expandingsch1_26`	2	3	7
`Expandingsch1_27`	2	3	7
`Expandingsch1_28`	3	5	14
`Expandingsch1_29`	3	5	14
`Expandingsch1_30`	3	5	14
`Expandingsch1_31`	1	1	3
`Expandingsch1_32`	1	1	3
`Expandingsch2_1`	1	1	N/A
`Expandingsch2_2`	1	1	3
`Expandingsch2_3`	1	2	N/A
`Expandingsch2_5`	1	2	N/A
`Expandingsch2_6`	1	2	N/A
`Expandingsch2_7`	1	1	3
`Expandingsch2_8`	1	1	3
`Expandingsch2_9`	1	2	N/A
`Expandingsch3_3`	2	2	4
`Expandingsch3_5`	2	2	4
`Expandingsch3_6`	2	2	4
`Expandingsch3_7`	2	2	N/A
`Expandingsch3_8`	2	2	4
`Expandingsch3_9`	2	2	5
`Expandingsch3_10`	2	N/A	N/A
`Expandingsch6_3`	2	N/A	N/A
`Expandingsch6_4`	2	N/A	N/A
`Expandingsch7_2`	2	N/A	N/A
`Expandingsch8_4`	2	N/A	N/A
`Expandingsch8_5`	1	2	N/A
`Expandingschema_9`	3	8	N/A

verification times in one column. The first time is a verification time with no theorem and the second one is a time with one theorem. There are three specifications that have many schema calculus definitions: `expandingschema_4`, `expandingschema_8`, and `expandingschema_9`. "`N/A`"s in several rows mean that the related specification cannot be translated by Z2SAL. All of these specifications contain re-declarations of state variables. It is because these variables could not be collapsed by our system to references of a state schema.

Regarding size of Z specifications, this will be discussed below. Tables 6, and 7 show us sizes of our Z specifications on this experiments. As can be seen from these three tables, a range of sizes of our Z specification inputs is between 1 and 3 kilobytes, otherwise the ranges are 1 to 8 and 1 to 14 for Z specification outputs and SAL specifications respectively. Sizes of SAL specifications shown in this table are original sizes producing by Z2SAL. As discussed above, a few of these SAL specifications have been modified as required in order to be executed by the SAL tool successfully or have been simplified to their compact form of predicates. Thus, their sizes can be different from the original ones.

"`input`" means a Z specification input file for our system. On the other hand, "`output`" means a Z specification output file generated by our system after performing an expansion process, "N/A"s in `output` means an associated Z specification input could not be expanded by our system either because of errors in the input file or because of bugs on our system, "N/A"s in `SAL specifications` means an associated Z specification could not be translated by Z2SAL. It can also be seen that a "N/A" in `input` makes this Z specification is not possible to be further processed.

4 Conclusion and Future Work

Our experiments find that the SAL language is not a case sensitive language. Another finding is that a bug on a Z2SAL translation of a range of numbers is found. This finding convinces us to such a bug since our other experiments with Z2SAL also find this.

All tables, which summarize our experiments, show that majority of our running examples can be redefined or expanded by our system. Several of them can also be translated by Z2SAL, verified by the SAL model checker, or simulated by the SAL simulator.

As a conclusion, redefinition and schema expansion, which pre-process a Z specification, can benefit the scope of translation of Z2SAL. It is because a Z specification can consist of generic constant or schema calculus definitions. This fact can support Z2SAL to translate a variety of Z specifications, which at the end can also support model checking Z specifications.

However, our method of implementing this system seems that our method is not feasible for larger and more complex specifications. It is because such specifications require more time to be translated by Z2SAL and to be executed by the SAL tool.

Expanded schemas can make the larger specification even larger. A more complex generic constant definition means several conditions. It can be more complex types of generic constant variables. It can also be a more complex predicate part of this definition. On the other hand, a more complex schema calculus definition means the definition contains a combination of several schema operators. Inevitably, further work could extend the system so it is able to run more complex Z specifications.

Furthermore, the out of memory error which is often encountered during simulation is also beneficial to be addressed. How abstraction can be applied to the related schemas to reduce the memory consumption is planned to be investigated.

Moreover, re-declaring state or global variables could be approached by implementing a better simplification in predicates. It could also include upgrading Z2SAL to a version that accepts many state schemas and references to them. Thus, one state schema can be specified to have just a variable part. Another state schema has a predicate part. Having these state schemas, a user can collapse state variables easily without a bother on negated predicates of other state schema.

Other future work is our approach to a SAL translation of a user defined function or constant could also be automated. There are two options for this automation: implementing it as an extension to this system or adding it as an extension to Z2SAL system. It appears that the second option is an easier method to implement.

Acknowledgment. The author would like to thank John Derrick, Siobhán North and Anthony J.H. Simons for giving the author a chance to work with their Z2SAL, discussions in this tool, and supervision in this study. A lot of thanks are dedicated to ISIHEMORA The Republic of Indonesia for its financial support of this study.

References

1. Jackson, D.: Abstract model checking of infinite specifications. In: FME 1994: Industrial Benefit of Formal Methods, pp. 519–531. Springer (1994)
2. Potter, B., Till, D., Sinclair, J.: An Introduction to Formal Specification and Z. Prentice Hall PTR, Upper Saddle River (1996)
3. West, M.M.: Issues in Validation and Executability of Formal Specifications in the Z Notation. Thesis of University of Leeds (2002)
4. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall Inc, Upper Saddle River (1996)
5. Derrick, J., North, S., Simons, A.J.H.: Z2SAL: a translation-based model checker for Z. Formal Asp. Comput. **23**(1), 43–71 (2011). Springer
6. Malik, P., Groves, L., Lenihan, C.: Translating Z to alloy. In: ASM, Alloy, B and Z, pp. 377–390. Springer (2010)
7. Barden, R., Stepney, S., Cooper, D.: Z in Practice. Prentice-Hall Inc, Upper Saddle River (1995)
8. Spivey, J.M.: The Z Notation. Prentice Hall, New York (1989)
9. Klein, G.: JFlex - The Fast Scanner for Java (2015). Accessed from http://www.jflex.de/index.html
10. Hurka, T.: BYACC/J (2008). Accessed from http://byaccj.sourceforge.net

11. Deitel, H., Deitel, P.J.: Java: How to Program, 5th edn. Prentice-Hall, Upper Saddle River (2003)
12. Siregar, M.U.: Support for model checking Z specifications. In: IEEE 17th International Conference on Information Reuse and Integration (IRI), pp. 241–248 (2016)
13. Plagge, D., Leuschel, M.: Validating Z specifications using the ProB animator and model checker. In: Integrated Formal Methods, pp. 480–500. Springer (2007)
14. Bolton, C.: Using the alloy analyzer to verify data refinement in Z. Electron. Notes Theoret. Comput. Sci. **137**(2), 23–44 (2005). Elsevier
15. De Moura, L., Owre, S., Shankar, N.: The SAL language manual. Computer Science Laboratory, SRI International, Menlo Park, CA, Technical report, CSL-01-01 (2003)
16. Smith, G., Wildman, L.: Model checking Z specifications using SAL. In: ZB 2005: Formal Specification and Development in Z and B, pp. 85–103. Springer (2005)
17. Derrick, J., North, S., Simons, A.J.H.: Issues in implementing a model checker for Z. In: Formal Methods and Software Engineering, pp. 678–696. Springer (2006)
18. Simons, A.J.H.: The Z2SAL User Guide (2012). Accessed from http://staffwww.dcs.shef.ac.uk/people/A.Simons/z2sal/userguide.html
19. Bensalem, S., Lakhnech, Y., Owre, S.: Computing abstractions of infinite state systems compositionally and automatically. In: Computer Aided Verification, pp. 319–331. Springer (1998)
20. King, P.: Printing Z and Object-Z LaTeXdocuments. University of Queensland, Department of Computer Science (1990)
21. Rann, D., Turner, J., Whitworth, J.: Z: a Beginner's Guide. CRC Press, Boca Raton (1994)
22. Marris, T.: Z notes (2007)
23. Hayes, I., Flinn, B.: Specification Case Studies. Prentice-Hall International, London (1987)

Reasoning About Temporal Faults Using an Activation Logic

André Didier(✉) and Alexandre Mota

Cidade Universitária, Av. Jornalista Anibal Fernandes,
s/n, Recife, PE 50740-560, Brazil
alrd@cin.ufpe.br

Abstract. Faults modelling is essential to anticipate failures in critical systems. Traditionally, Static Fault Trees (SFTs) are employed to this end, but Temporal and Dynamic Fault Trees (TFTs and DFTs) are gaining evidence due to their enriched power to model and detect intricate propagation of faults that lead to a failure. SFTs structure can be abstracted to Boolean expressions. An algebra with an operator to express order is needed to abstract TFT and DFT structures. These expressions for SFT, TFT, and DFT are called structure expressions.

Architectural modelling languages, such as Architecture and Analysis Design Language (AADL), have been used to model components and systems relations, including modelling of faults, errors, failures, and fault propagation. AADL tools can perform Static Fault Tree Analysis, for the faults modelled using AADL's Error Model Annex.

In previous work, we showed an Algebra of Temporal Faults to analyse the order of occurrence of faults extending Boolean algebra to perform analysis for Temporal and Dynamic fault trees. In this work, we show a parametrized logic to express nominal and erroneous behaviours, including faults modelling, provided an algebra and a set of operational modes. We show how to use this logic together with the Algebra of Temporal Faults to analyse the occurrence of faults as well as their order and propagation. The logic created in this work is intended to help analysts to consider all possible situations in complex expressions with order-related operators, avoiding to miss some subtle (but relevant) combination.

Keywords: Activation logic · Algebra of temporal faults · Dynamic fault trees · Boolean algebra

1 Introduction

The development process of critical control systems is based essentially on the rigorous execution of guides and regulations [1,4,10,11]. Specialised agencies (like FAA, EASA and ANAC in the aviation field) use these guides and regulations to certify such systems.

Safety plays a crucial role on critical systems and it is the responsibility of the safety assessment process. ARP-4761 [1] defines several techniques to

© Springer International Publishing AG 2018
S.H. Rubin and T. Bouabana-Tebibel (eds.), *Quality Software Through Reuse and Integration*, Advances in Intelligent Systems and Computing 561,
DOI 10.1007/978-3-319-56157-8_13

perform safety assessment. One of them is Fault Tree Analysis (FTA). It is a deductive process that uses trees to model faults and their dependencies and propagation. In such trees, the premises are the leaves (basic events) and the conclusions are the roots (top events). Intermediary events use gates to combine basic events and each kind of gate has its own combination semantics definition. For example, the most traditional gates are OR and AND. They combine the events as *at least one shall occur* and *all shall occur*, respectively. To analyse fault trees, their structures are then abstracted as Boolean expressions called *structure expressions*. The analysis with these two traditional gates uses a well-defined algorithm based on Shannon's method—which originated the Binary Decision Diagrams (BDDs) [3,6]—to obtain minimal cut sets from the structure expressions and a general formula to calculate the probability of top events.

Besides the traditional OR and AND gates, the Fault Tree Handbook defines other gates. For example the Priority-AND gate, which considers the order of occurrence of events. Although the work reported in [24] defines these new gates, there is no algorithm to perform the analysis of trees that contain such new gates. This motivated the introduction of two new kinds of fault trees: Dynamic Fault Trees [9] (DFTs) and Temporal Fault Trees [26,27] (TFTs). These variant trees can capture sequence dependencies of fault events in a system. The difference from Temporal Fault Tree [26,27] (TFT) to Dynamic Fault Tree [9] (DFT) is that Temporal Fault Trees [26,27] (TFTs) use temporal gates directly, while Dynamic Fault Trees [9] (DFTs) do not—Dynamic Fault Trees [9] (DFTs) gates are an abstraction of temporal gates. To differentiate traditional fault trees from the other two, we will call traditional fault trees as Static Fault Trees (SFTs).

Both TFT and DFT also use structure expressions ([19,27], respectively) to abstract the tree to enable their analyses. Despite some limitations related to spare gates [19], the structure expressions used in TFTs and DFTs can be formulated in terms of a generic order-based operator.

The NOT operator is absent in the algebras showed in [16,18,25,27]. They conceptually remove such an operator to avoid incorrect analysis, as there is no consensus about the relevance of its use: (i) it can be misleading, generating non-coherent analysis [22], or (ii) it can be essential in practical use [5]. The algebra created in our previous work [8] defines the NOT operator and allows its use.

In structure expressions, the variables represent fault events and the expressions represent a top event, an operational mode of a system. The combination of all operational modes is expected to describe the complete behaviour of a system. For example, if no fault occurs, then the system is in a nominal state; if all faults occur, then the system is definitely in a faulty state. Possibilities in between vary accordingly to the fault tolerance strategies employed in a system. The analysis of all possibilities is what we call *completeness analysis*. For Boolean algebra, it is equivalent to verify if all rows in a truth table (in which the variables are fault events) are considered in at least one structure expression.

Architecture Analysis and Design Language [12] (AADL) is a standard language to model (among other features) system structure and component interaction. AADL has several tools to perform different analyses to obtain SFT to perform FTA. But AADL's assertions framework does not express order explicitly as needed for TFT and DFT analyses.

On the analyses of systems and its constituents, there is a distinction of operational modes and error events. Operational modes refer to the behaviour that is perceived on the boundaries of a system. Error events, on the other hand, represent the behaviour detected in a constituent of a system. Such error events may relate to an operational mode, but not necessarily. We abstract these differences and leave the distinction as a parameter. In this article, we refer to such a set as *operational modes*.

Another concern, left untreated in the literature, is the undesirable possibility of non-determinism in structure expressions. What guarantees can we provide to detect non-determinism in erroneous behaviour? For example, if a commission is observed when fault A is active and an omission is observed when faults A and B are active, then the system may behave non-deterministically with a commission or omission when both A and B are active (A and B implies A). In this work we show three different approaches to check the non-determinism: (i) verify its existence, (ii) indicate which set of operational modes are active for a combination of faults, or (iii) what is the combination of faults that activates a set of operational modes.

Writing and analysing expressions with order-related operators is more complex than analysing expressions with Boolean operators only. In this work, we define a formal Activation Logic (AL) that works together with an inner algebra to perform analysis of system structure and component interaction with a focus on fault modelling and fault propagation, tackling the complexity introduced by order-related operators. AL receives an algebra and the set of operational modes of a system as parameters. The choice of algebra defines which structure expressions can be obtained: if Boolean algebra is passed as a parameter, the AL can generate structure expressions with Boolean operators (SFT); if the Algebra of Temporal Faults [8] (ATF) is passed as a parameter, the AL can generate structure expressions with order-related operators (TFT and DFT). The AL requires that the inner algebras provide a set of properties (tautology and contradiction) and semantic values. The use of the NOT is essential: besides its use in expressions, we use the complement to normalise the expressions to provide *healthy* expressions. To obtain critical event expressions used in FTs and to denote faults propagation, the AL provides a predicates notation and verification of non-determinism.

This paper is organised as follows: in Sect. 2 we show the concepts used as the basis for this work. Section 3 presents the proposed AL, and Sect. 4 the case study and the application of the proposed AL. Finally, we present our conclusions and future work in Sect. 5.

2 Background

Faults modelling depends on which analyses we want to perform. For instance, in fault trees, even if a fault can be repaired, it is considered as a nonrepairable fault. A fault tree is a snapshot[1] of a system's, subsystem's or component's topology of faults. The time relations on fault events in TFTs and DFTs allows the analysis of different configurations (or snapshots) of a system, subsystem or component. We discuss these time relations in Sect. 2.1.

Structure expressions are used to analyse fault trees. In general, a structure expression is obtained from gates semantics and basic events. Basic events become variables and gates become operators (a gate may become one or more operators). In Sect. 2.2 we explain these structure expressions for SFTs, TFTs and DFTs.

The AL proposed in this work depends on algebraic rules and relies on a *complement* operator. Our previous work showed the ATF that extends the Boolean algebra, thus providing the NOT operator and some properties and rules to use the algebra. In Sect. 2.3 we show these properties and rules used in this work.

2.1 Time Relation of Fault Events

The most general case for time relations is to consider that each fault event has a continuous time duration. They are the basis on how fault events discretization are defined. The point of view in this work is the analysis of the effects caused by a combination of faults in a snapshot of a system state. In Fig. 1 we show all possibilities of events relations in a continuous timeline (from A to B; the converse relation is similar):

a. A starts before and ends after B has started, but before B has ended;
b. A starts before B and ends after B has ended (A contains B);
c. B starts after A, but they end at the same time;
d. A and B start at the same time, but A ends before B;
e. A and B start and end at the same time;
f. A starts before B and ends when B starts.
g. A starts and ends before B starts;

Considering that fault occurrence corresponds to the start of a fault event and its duration, from Fig. 1 we clearly identify which event comes first: A comes before B, except in the cases (d) and (e), where they start exactly at the same time. If fault events are independent (they are not susceptible to have a common cause) then the probability of their occurrences starting at the same time is very low. The relations (f) and (g) shows the case that the system was repaired, thus A is not active when B starts. In Sect. 2.3 we show that the ATF abstracts the relation of events in continuous time as an *exclusive before* relation, based on fault *occurrence* (it is similar—at least implicitly—to what is reported in [17,27]).

[1] Whether a top event indeed causes a catastrophic or major failure is out of the scope of this paper; we consider that, if it is possible that such failure occurs, then it will.

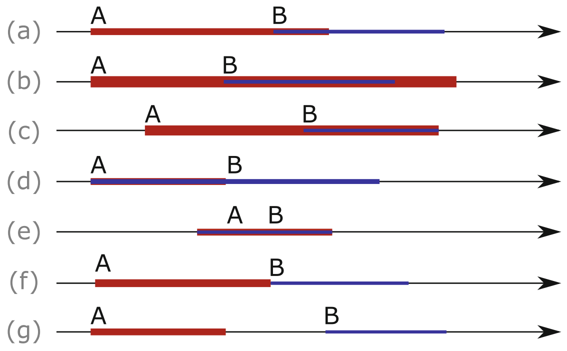


Fig. 1. Relation of two events with duration

2.2 Structure Expressions

Structure expressions in FTA are defined in terms of set theory, using symbols for fault events occurrence. If a fault event symbol is in a set, then it means that fault has occurred. A set is a combination of fault events that causes the top-level event of a tree. A structure expression of a tree is denoted by a set of sets of fault event combinations. The OR gate becomes the union operator between sets and the AND gate, the intersection. For example, if a system contains fault events a, b, and c, fault trees for this system contain at most all these three events. The occurrence of the fault event a is denoted by a set of sets A, which contains the following sets:

1. $\{a\}$: only a occurs;
2. $\{a, b\}$: a and b occur;
3. $\{a, c\}$: a and c occur;
4. $\{a, b, c\}$: all three events occur.

The fault tree in Fig. 2 contains only two events and the resulting structure expression for this tree is the expression $A \cap B$ (TOP), where A and B are the sets of sets that contain a and b, respectively. The resulting combinations for TOP are $\{a, b\}$ and $\{a, b, c\}$ (fault events a and b occur in all possibilities).

After obtaining structure expressions, the next step is to reduce the expressions to a canonical form to obtain the *minimal cut sets*. Minimal cut sets are the sets that contain the minimum and sufficient events to activate the top-level failure. That is, minimal cut sets are the smallest sets of fault events that, if all occur, cause the top-level failure to occur. Probabilistic analysis is then performed on these events to obtain the overall probability of occurrence of the top-level event. The Fault Tree Handbook shows an algorithm based on Shannon's method to reduce structure expressions to obtain minimal cut sets. The Boolean expression of the tree shown in Fig. 2 is $TOP = A \wedge B$.

Structure expressions are also present in TFTs [25,27,28], through the Pandora[2] methodology. These expressions use the FTA operators OR and AND,

[2] Pandora stands for: P-AND-ORA, which translates to Priority AND, Time.

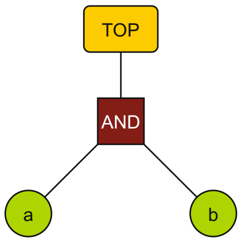


Fig. 2. Very simple example of a fault tree

and three new operators related to events ordering: Priority-AND (PAND), Priority-OR (POR), and Simultaneous-AND (SAND). The semantics of the PAND in TFTs is similar to the semantics of the Priority-AND described in the Fault Tree Handbook. To avoid ambiguous expressions, the semantics in TFTs is stated in terms of natural numbers (instead of Boolean values), using a *sequence value* function. For every possibility it assigns a sequence value to each fault event. For example, if event A occurs before event B, then the sequence value of A is lower than the sequence value of B, and one formula to express this is A PAND B.

In TFTs, an invariant on sequence values is that there are no gaps for assigned values higher than zero. For example, if faults A and B occur at the same time and there are only these two events, then they should both be assigned value 1. On the other hand, if A occurs before B, then the assigned values are 1 and 2, respectively. Value zero means that the event is not active in the combination. Table 1 shows the semantics of all TFT operators with sequence values.

Table 1. TFT operators and sequence value numbers

A	B	AND	OR	PAND	POR	SAND
0	0	0	0	0	0	0
0	1	0	1	0	0	0
1	0	0	1	0	1	0
1	1	1	1	0	0	1
1	2	2	1	2	1	0
2	1	2	1	0	0	0

The reduction of TFT expressions is achieved using dependency trees. In a dependency tree, if all children of a tree node are true, then the node is also true. Conversely, if a node is true, then all its children are also true. An issue with dependency trees is that they grow exponentially. Accordingly to the work reported in [28], it is already infeasible to deal with seven fault events in TFTs. They use an alternative solution based on modularisation and algebraic laws [27] to tackle this.

Structure expressions are also used in DFTs. In [16,17,20] fault events occur in a specific time and are instantaneous, stated through a *date-of-occurrence* function. As the date-of-occurrence function is stated in continuous time, the probability of two events occurring at the same time is negligible. In fact, useful information is obtained from the possibilities of relation in time of the occurrence of the events.

The work reported in [16,17,20] describe an algebra with operators OR and AND, and three new operators to express events ordering: (i) non-inclusive-before, (ii) simultaneous, and (iii) inclusive-before. The non-inclusive-before and the simultaneous operators are similar to TFT's POR and SAND operators, respectively (although in [16,17,20] the only *true* result with the simultaneous operator happens if the operands are the same). The inclusive-before is a composition of the non-inclusive-before and the simultaneous operators.

The work reported in [23,29] shows the top-level events probability calculation for DFTs by converting them to a simplified version, using only order-based operators. Such a simplified version, which is based on a modified BDD that includes an order-based operator, creates Sequential BDDs that are used to perform the probabilistic analysis.

From the previous explanation, we can conclude that an order-based operator is present on the analyses of TFTs and DFTs. Each approach describes a new algebra (without the NOT operator) based on different representations of events ordering with similar theorems to reduce expressions to a canonical form.

2.3 The Algebra of Temporal Faults

Recall from Sects. 2.1 and 2.2 that fault events are independent on one another if the events are not susceptible to a common cause. Also, the simultaneity of events is probabilistically impossible, so one event occurs exclusively before or after another one, inducing an order of occurrence of events. Moreover, the analysis of fault events considers that they have started and are active, as a snapshot of a system (faulty) state. Thus, the ATF is not used to analyse the effects of a repairable fault. For example, the cases that are possible to analyse with the theory shown here are (a), (b) and (c) of Fig. 1, in Sect. 2.1.

The set-theoretical abstraction of structure expressions for SFTs [24, pp. VI-11] is very close to a Free Boolean Algebra [13, pp. 256–266] (FBA), where each generator in FBA corresponds to a fault event symbol in fault trees. In FBAs, as generators are "free", they are independent on one another and Boolean formulas are written as a set of sets of possibilities, which are similar to the structure expressions of SFTs.

The set of sets for FBAs is the denotational semantics for Boolean algebras. We use the concept of generators to define the denotational semantics of ATF using a set of lists without repetition (distinct lists). The choice of lists is because this structure inherently associates a generator to an index, making implicit the representation of order. These lists are composed by non-repeated elements because the events in fault trees are non-repairable, thus they do not occur more than once.

In the following, we show the definitions and laws of the ATF used in Sect. 4. To avoid repetition, let S, T and U be sets of distinct lists. A list xs is distinct if it has no repeated element. So, if x is in xs, then it has a unique associated index i and we denote it as $x = xs_i$.

The ATF form a free algebra, similarly to FBAs. *Infimum* and *Supremum* are defined as set intersection ($\cap$) and union ($\cup$) respectively. The order within the algebra is defined with set inclusion ($\subseteq$).

To distinguish the permutations that are not defined in FBA, we need a new operator. The definition of XBefore ($\rightarrow$) is given in terms of list concatenation:

$$\begin{aligned} S \rightarrow T \ &= \{zs | \exists xs, ys \bullet (\mathbf{set}\, xs) \cap (\mathbf{set}\, ys) = \{\} \\ &\quad \wedge xs \in S \wedge ys \in T \wedge zs = xs@ys\} \end{aligned} \tag{1}$$

where the **set** function returns the set of the elements of a list, and @ concatenates two lists.

In some cases it is more intuitive to use the XBefore definition in terms of lists slicing because it uses indexes explicitly. Lists slicing is the operation of taking or dropping elements, obtaining a sublist. In slicing, the starting index is inclusive, and the ending is exclusive. Thus the first index is 0 and the last index is the list length. For example, the list $xs_{[i..|xs|]}$ is equal to the xs list, where $|xs|$ is the list length. We use the following notation for list slicing:

$$xs_{[i..j]} = \text{starts at } i \text{ and ends at } j - 1 \tag{2a}$$

$$xs_{[..j]} = xs_{[0..j]} \tag{2b}$$

$$xs_{[i..]} = xs_{[i..|xs|]} \tag{2c}$$

List slicing and concatenation are complementary: concatenating two consecutive slices results in the original list:

$$\forall i \bullet xs_{[..i]}@xs_{[i..]} = xs \tag{3}$$

There is an equivalent definition of XBefore with concatenation using lists slicing:

$$S \rightarrow T = \{zs | \exists i \bullet zs_{[..i]} \in S \wedge zs_{[i..]} \in T\} \tag{4}$$

A variable in ATF is defined by one generator, and denotes its occurrence:

$$\mathbf{var}\, x = \{zs | x \in zs\} \tag{5}$$

The following expressions are sufficient to define the ATF in terms of an inductively defined set (**atf**):

$$\mathbf{var}\, x \in \mathbf{atf} \qquad \text{Variable} \tag{6a}$$

$$S \in \mathbf{atf} \implies -S \in \mathbf{atf} \qquad \text{Complement, Negation} \tag{6b}$$

$$S \in \mathbf{atf} \wedge T \in \mathbf{atf} \implies S \cap T \in \mathbf{atf} \qquad \text{Intersection, } \textit{Infimum} \tag{6c}$$

$$S \in \mathbf{atf} \wedge T \in \mathbf{atf} \implies S \rightarrow T \in \mathbf{atf} \qquad \text{XBefore} \tag{6d}$$

Following the definitions, the expressions below are also valid for **atf** (using DeMorgan laws):

$$UNIV \in \textbf{atf} \qquad \text{Universal set, True} \tag{6e}$$

$$\{\} \in \textbf{atf} \qquad \text{Empty set, False} \tag{6f}$$

$$S \in \textbf{atf} \wedge T \in \textbf{atf} \implies S \cup T \in \textbf{atf} \qquad \text{Union, } \textit{Supremum} \tag{6g}$$

The following expressions are valid for generators a and b and are sufficient to show that the generators are independent:

$$\textbf{var}\, a = \textbf{var}\, b \iff a = b \tag{7a}$$

$$\textbf{var}\, a \nsubseteq -\textbf{var}\, b \tag{7b}$$

$$\textbf{var}\, a \neq -\textbf{var}\, b \tag{7c}$$

$$-\textbf{var}\, a \nsubseteq \textbf{var}\, b \tag{7d}$$

$$-\textbf{var}\, a \neq \textbf{var}\, b \tag{7e}$$

Expressions (6a) to (6g) and (7a) to (7e) imply that the ATF without the XBefore operator (1) forms a Boolean algebra based on sets of lists. And this is also equivalent to an FBA with the same generators.

Note that, as the ATF is a conservative extension of a Boolean algebra, the NOT operator is defined here, so expressions in the ATF can use it.

In the following section, we show properties as a generalisation of the preconditions of laws related to XBefore.

Temporal properties (*tempo*). Temporal properties give a more abstract and less restrictive shape on the XBefore laws. These properties avoid the requirement that every operand of XBefore should be a variable (5).

The first temporal property is about disjoint split. If the first part of a list is in a given set, then every remainder part is not. So, if a generator is in the beginning of a list, it must not be at the ending (and vice-versa).

$$\textbf{tempo}_1 S = \forall i, j, zs \bullet i \leq j \implies \neg\left(zs_{[..i]} \in S \wedge zs_{[j..]} \in S\right) \tag{8a}$$

$$\textbf{tempo}_2 S = \forall i, zs \bullet zs \in S \iff zs_{[..i]} \in S \vee zs_{[i..]} \in S \tag{8b}$$

$$\textbf{tempo}_3 S = \forall i, j, zs \bullet j < i \implies \left(zs_{[j..i]} \in S \iff zs_{[..i]} \in S \wedge zs_{[j..]} \in S\right) \tag{8c}$$

$$\textbf{tempo}_4 S = \forall zs \bullet zs \in S \iff (\exists i \bullet zs_{[i..(i+1)]} \in S) \tag{8d}$$

The second temporal property is about belonging to one sublist in the beginning or in the end. If a generator is in a list, then it must be at the beginning or at the ending.

The third temporal property is about belonging to one sublist in the middle. If a generator belongs to a sublist between i and j, then it belongs to the sublist that starts at first position and ends in j and to the sublist that starts at i and ends at the last position (both sublists contain the sublist in the middle).

Finally, if a generator belongs to a list, then there is a sublist of size one that contains the generator.

Variables have all four temporal properties. For a generator x, the following is valid:

$$\mathbf{tempo}_1 (\mathbf{var}\, x) \wedge \mathbf{tempo}_2 (\mathbf{var}\, x) \wedge \mathbf{tempo}_3 (\mathbf{var}\, x) \wedge \mathbf{tempo}_4 (\mathbf{var}\, x)$$

Other expressions also meet one or more temporal properties:

$$\mathbf{tempo}_1 S \wedge \mathbf{tempo}_1 T \implies \mathbf{tempo}_1 (S \cap T) \tag{9a}$$
$$\mathbf{tempo}_3 S \wedge \mathbf{tempo}_3 T \implies \mathbf{tempo}_3 (S \cap T) \tag{9b}$$
$$\mathbf{tempo}_2 S \wedge \mathbf{tempo}_2 T \implies \mathbf{tempo}_2 (S \cup T) \tag{9c}$$
$$\mathbf{tempo}_4 S \wedge \mathbf{tempo}_4 T \implies \mathbf{tempo}_4 (S \cup T) \tag{9d}$$

XBefore Laws. We now show some laws to be used in the algebraic reduction of ATF formulas. The laws follow from the definition of XBefore, from events independence, and from the temporal properties.

We define events independence ($\lhd\!\rhd$) as the property that one operand does not imply the other. For example, we need to avoid that the operands of XBefore are $\mathbf{var}\, a$ and $\mathbf{var}\, a \cup \mathbf{var}\, b$ (it results in $\{\}$, see (11e)).

$$S \lhd\!\rhd T = \forall i, zs \bullet \neg \left(zs_{[i..(i+1)]} \in S \wedge zs_{[i..(i+1)]} \in T \right) \tag{10}$$

The absence of occurrences ($\{\}$, the empty set of **atf**) is a "0" for the XBefore operator.

$$\{\} \rightarrow S = \{\} \qquad \text{left-false-absorb} \tag{11a}$$
$$S \rightarrow \{\} = \{\} \qquad \text{right-false-absorb} \tag{11b}$$
$$(S \rightarrow T) \cup S = S \qquad \text{left-union-absorb} \tag{11c}$$
$$(T \rightarrow S) \cup S = S \qquad \text{right-union-absorb} \tag{11d}$$
$$\mathbf{tempo}_1 S \implies S \rightarrow S = \{\} \qquad \text{non-idempotent} \tag{11e}$$
$$\mathbf{tempo}_1 S \wedge \mathbf{tempo}_1 T \wedge \mathbf{tempo}_1 U \implies S \rightarrow (T \rightarrow U) = (S \rightarrow T) \rightarrow U \qquad \text{associativity} \tag{11f}$$

The XBefore is absorbed by one of the operands: if one of the operands may happen alone, thus the order with any other operand is irrelevant. However, an event cannot come before itself, thus XBefore is not idempotent Finally, the XBefore is associative.

To allow formula reduction we need the relation of XBefore to the other Boolean operators. We use the XBefore as operands of union and intersection.

$$\mathbf{tempo}_1 S \wedge \mathbf{tempo}_1 T \implies (S \rightarrow T) \cap (T \rightarrow S) = \{\} \qquad \text{inter-equiv-false} \tag{12a}$$
$$\mathbf{tempo}_{1-4} S \wedge \mathbf{tempo}_{1-4} T \wedge S \lhd\!\rhd T \implies (S \rightarrow T) \cup (T \rightarrow S) = S \cap T \qquad \text{union-equiv-inter} \tag{12b}$$

As the XBefore is not symmetric, the intersection of symmetrical sets is empty. The union of the symmetric is a partition of the intersection of the operands.

There are other laws shown in [8]. We are still working on a syntactical reduction for tautology and contradiction. Such an analysis for Boolean algebra uses binary trees for formula reduction. Our initial studies show that the ATF relies on a ternary tree. Besides such a syntactical analysis, we only need those laws shown in this section.

3 The Activation Logic

The Activation Logic (AL) proposed in this work emerges from the need to analyse the behaviour of a system when a subset of the faults is active during the same time period, and to provide completeness analysis of system behaviour. There are at least two strategies to use AL to obtain structure expressions of SFT, TFT, or DFT: (i) model systems directly in AL, and (ii) obtaining operational mode expressions extracted from failure traces, as shown in the work reported in [8]. In approaches as those reported in [16,27], behavioural completeness is left for the analyst. Using tautology and the indication of undefined nominal values, we ensure that no situation is left forgotten.

The AL associates: (i) an operational mode, and (ii) the expression of fault events that *activates* the operational mode or error event. The expressions of fault events can be written in any algebra that provides tautology and contradiction properties. Thus, AL is parametrized by: (i) an algebra that provides at least tautology and contradiction, and (ii) operational modes. Figure 3 depicts an overview of AL.

We summarise the properties of the AL as follows:

1. No expression predicate is a contradiction: there are no *false* predicates in activation expressions;
2. The predicates in the terms of an expression consider all possible situations: expression tautology;
3. There are no two terms with exactly the same operational mode: all expression terms are related to a unique operational mode.

These properties form the *healthiness conditions* [14] of an expression in the AL.

We show the general form of the AL to model faults in Sect. 3.1, the healthiness conditions to normalize expressions in Sect. 3.2, how to identify non-determinism in an expression in Sect. 3.3, and the predicates notation to analyse systems and model fault propagation in Sect. 3.4.

3.1 The Activation Logic Grammar

Each term in an expression is a pair of a predicate and an operational mode. The predicate is written in either Boolean algebra, ATF, or any algebra that provides these properties: tautology and contradiction. We assume that the set

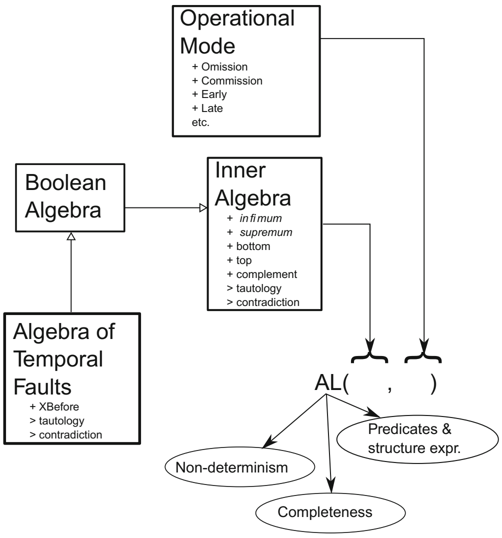


Fig. 3. Activation Logic (AL) overview

of possible faults on a system is finite and that each variable declared in a predicate represents a fault event.

The operational mode has two generic values: (i) Nominal, and (ii) Failure. Nominal values either determine value, or an undefined value (in this case, the constant value "$undefined$" is assumed). Failure values denote an erroneous behaviour, which can be a total failure (for example, signal omission) or a failure that causes degradation (for example, a signal below or above its nominal range). The choice of the operational modes depends on the system being analysed and its definition is generic and is left for the analyst. For the AL, it is sufficient to specify that it is an erroneous behaviour.

The grammar is parametrized by the syntax of an algebra (`Algebra`) and a set of operational modes (`OperModes`). The initial rules of the grammar are defined as follows:

```
AL(Algebra, OperModes)      = TERM(Algebra, OperModes)
                              | TERM(Algebra, OperModes)
                                '|' AL(Algebra, OperModes)
TERM(Algebra, OperModes)    = '(' Algebra ',' OM(OperModes) ')'
OM(OperModes)               = 'Nominal' NominalValue
                              | 'Failure' OperModes
NominalValue                = 'undefined' | Number
Number                      = Integer | Bool | Decimal
```

The denotational semantics of the expressions in AL is a set of pairs. The predicate in each term of an expression depends on the semantics of the inner algebra. Thus the predicate evaluates to either *true* ($\top$) or *false* ($\bot$) depending on the valuation in the algebra. In what follows we show a sketch of the denotational semantics of AL.

$$\begin{aligned}
(\mathtt{P}_1, \mathtt{O}_1) &\mapsto \{(P_1, O_1)\} \\
(\mathtt{P}_1, \mathtt{O}_1) \,|\, (\mathtt{P}_2, \mathtt{O}_2) &\mapsto \{(P_1, O_1), (P_2, O_2)\} \\
\mathtt{Nominal\ 100} &\mapsto \text{Nominal}\, 100 \\
\mathtt{Nominal\ undefined} &\mapsto \text{Nominal}\, undefined \\
\mathtt{Failure\ Omission} &\mapsto \text{Failure}\, Omission
\end{aligned}$$

In an expression, if the ith predicate evaluates to *true* ($\top$), we say that the ith operational mode is *activated*. To simplify the presentation of the expressions and to ease the understanding, we use the denotational semantics in the remainder of this article (the right-hand side of the sketch above).

In this section, to illustrate the properties and possible analyses, we use an example of a system with faults A and B and the following outputs:

O_1**:** when A is active;
O_2**:** when B is active;
O_3**:** when A is active, but B is not;
O_4**:** when A or B are active.

The expression for this example in AL is:

$$S = \{(A, O_1), (B, O_2), (A \land \neg B, O_3), (A \lor B, O_4)\} \tag{13}$$

In this example we see that one of the healthiness conditions is not satisfied: when for instance, A and B are both inactive ($\neg (A \land B)$), there is no explicit output defined. In Sect. 4 we show a more detailed case study to illustrate the reasoning about temporal faults. In the next section, we show how to normalise the expression, so that the three healthiness conditions are satisfied.

3.2 Healthiness Conditions

The healthiness conditions are fix points of a language. The property is defined as a function of an expression and returns another expression. For example, if a healthiness condition H is satisfied for an expression Exp, thus $\text{H}(Exp) = Exp$.

In what follows we show the three healthiness conditions for the AL. All definitions in this section refer to an algebra that has the following properties:

contradiction: the expression always evaluates to *false*;
tautology: the expression always evaluates to *true*.

$\mathbf{H_1}$**: No predicate is a contradiction.** This property is very simple and it is used to eliminate any term that has a predicate that always evaluates to false.

Definition 1. *Let exp be an expression in the AL, then:*

$$\text{H}_1(exp) = \{(P, O) \,|\, (P, O) \in exp \bullet \neg\text{contradiction}(P)\} \tag{14}$$

where the operator $\in$ indicates that a term is present in the expression.

Applying the first healthiness condition to our example results in:

$$\mathrm{H}_1(S) = S$$

Thus, we conclude that S *is* H_1-healthy.

H_2: All possibilities are covered. This property is used to make explicit that there are uncovered operational modes. In this case, there is a combination of variables in the inner algebra that was not declared in the expression. Very often the focus when modelling faults is the erroneous behaviour, so we assume that such an uncovered operational mode is nominal, but has an undefined value.

Definition 2. *Let exp be an expression in the AL, and τ is:*

$$\tau = \neg\left(\bigvee_{(P,O)\in exp} P\right)$$

then:

$$\mathrm{H}_2(exp) = \begin{cases} exp, & \textit{if}\ \text{contradiction}(\tau) \\ exp \cup \{(\tau, \text{Nominal}\ undefined)\}, & \textit{otherwise} \end{cases} \tag{15}$$

This property states that if the expression is already complete, so all possibilities are already covered, thus the expression is healthy.

Applying the second healthiness condition to our example results in the following expression after simplification:

$$\mathrm{H}_2(S) = S \cup \{(\neg A \wedge \neg B, \text{Nominal}\ undefined)\}$$

Thus, we conclude that S *is not* H_2-healthy.

H_3: There are no two terms with exactly the same operational mode. This property merges terms that contain the same operational mode. It avoids unnecessary formulas and may reduce the expression.

Definition 3. *Let exp be an expression in the AL. Then:*

$$\begin{aligned} \mathrm{H}_3(exp) = \{\ & (P_1, O_1) \mid (P_1, O_1) \in exp \wedge \\ & \forall (P_2, O_2) \in exp \bullet (P_1, O_1) = (P_2, O_2) \vee O_1 \neq O_2\ \} \cup \\ & \{(P_1 \vee P_2, O_1) \mid (P_1, O_1), (P_2, O_2) \in exp \wedge O_1 = O_2\} \end{aligned} \tag{16}$$

Applying H_3 in the example in the beginning of the section, we conclude that S *is* H_3-healthy. On the other hand, if we consider an S' system being a copy of S, but making $O_1 = O_2$, then:

$$\mathrm{H}_3(S') = \{(A \vee B, O_1), (A \wedge \neg B, O_3), (A \vee B, O_4)\}$$

Thus, we conclude that S' *is not* H_3-healthy.

Healthy Expression. To obtain a healthy expression, we apply all three healthiness conditions. The order of application of each healthiness condition does not change the resulting expression. The healthiness function is written as composition of functions as follows:

$$\mathrm{H} = \mathrm{H}_1 \circ \mathrm{H}_2 \circ \mathrm{H}_3 \tag{17}$$

After applying the three healthiness conditions to S, the resulting expression is:

$$\begin{aligned}\mathrm{H}(S) = \{\ & (A, O_1), (B, O_2), \\ & (A \wedge \neg B, O_3), (A \vee B, O_4), \\ & (\neg A \wedge \neg B, \mathrm{Nominal}\, undefined)\ \}\end{aligned}$$

The healthiness conditions are useful to faults modelling, aiding the faults analyst to check contradictions and completeness. Also, obtaining safe predicates is only possible in healthy expressions. In the next section, we show how to verify non-determinism in AL expressions.

3.3 Non-determinism

The non-determinism is usually an undesirable property. It causes an unexpected behaviour, so the analysis shall consider the activation of fault even if the fault might or not be active.

To identify a non-determinism, we can check for the negation of a contradiction in a pair of predicates in the inner algebra.

Definition 4 (Non-determinism). *Let exp be an expression in AL.*

$$\begin{aligned}\mathrm{nondeterministic}\,(exp) =& \exists\,(P_1, O_1), (P_2, O_2) \in exp\ \bullet \\ & \neg \mathrm{contradiction}\,(P_1 \wedge P_2)\end{aligned} \tag{18}$$

If there is at least one combination that evaluates $P_1 \wedge P_2$ to true (it is not a contradiction), then *exp* is non-deterministic. Our example is clearly non-deterministic as at least $A \wedge (A \vee B)$ is not a contradiction.

To analyse components and systems, and to model faults propagation, a predicates notation is shown in the next section. The predicates notation offers more two ways to check non-determinism.

3.4 Predicates Notation

The AL needs a special notation to enable the analysis of: (i) a particular faults expression, or (ii) a propagation in components. Such a special notation extracts predicates in the inner algebra given an output of interest.

Definition 5 (Predicate). *Let exp be an expression in AL, and O_x an operational mode. A predicate over exp that matches O_x is then:*

$$\langle \mathrm{out}\,(exp) = O_x \rangle \iff \exists\,(P, O) \in \mathrm{H}\,(exp)\ \mid\ O = O_x \bullet P \tag{19}$$

The predicate notation function returns a predicate in the inner algebra. For the example in the beginning of this section, the predicate for O_2 is obtained as follows:

$$\langle \text{out}\,(S) = O_2 \rangle = B$$

To allow fault propagation of components we need another special notation that expands the modes of an expression with a predicate in the inner algebra.

Definition 6 (Modes). *Let exp be an expression in AL, and P a predicate in the inner algebra, then:*

$$\text{modes}\,(exp, P) = \{(P_i \wedge P, O_i) \;\mid\; (P_i, O_i) \in \text{H}\,(exp)\} \tag{20}$$

Finally, to check the possible outputs, we need a function to obtain a set of outputs given an expression.

Definition 7 (Activation). *Let exp be an expression in AL, and P_x a predicate in the inner algebra, then:*

$$\text{activation}\,(exp, P_x) = \{O | (P, O) \in \text{H}\,(exp) \wedge \text{tautology}\,(P_x \implies P)\} \tag{21}$$

The non-determinism can also be checked using the predicates notation and the activation property:

$$\text{activation}\,(S, A \wedge \neg B) = \{O_1, O_3\} \tag{22a}$$

$$\langle \text{out}\,(S) = O_1 \rangle \wedge \langle \text{out}\,(S) = O_3 \rangle = \; A \wedge \neg B \tag{22b}$$

Equation (22a) shows that both O_1 and O_3 can be observed if $A \wedge \neg B$ is *true*. Equation (22b) states that if the possible operational modes of healthy S are O_1 and O_3, then the predicate is $A \wedge \neg B$. In the next section, we show a practical case study using these properties and notations.

4 Case Study

EMBRAER provided us with the Simulink model of an Actuator Control System (depicted in Fig. 4). The failure logic of this system (that is, for each of its constituent components) was also provided by EMBRAER (we show some of them in Table 2). In what follows we illustrate our strategy using the Monitor component.

A monitor component is a system commonly used for fault tolerance [15,21]. Initially, the monitor connects the main input (power source on input port 1) with its output. It observes the value of this input port and compares it to a threshold. If the value is below the threshold, the monitor disconnects the output from the main input and connects to the secondary input. We present the Simulink model for this monitor in Fig. 5.

Now we show two contributions: (i) using only Boolean operators, thus ignoring ordering, we can obtain the same results obtained in [7], and (ii) using the order-related operator reported in [8] obtaining an expression in ATF with the

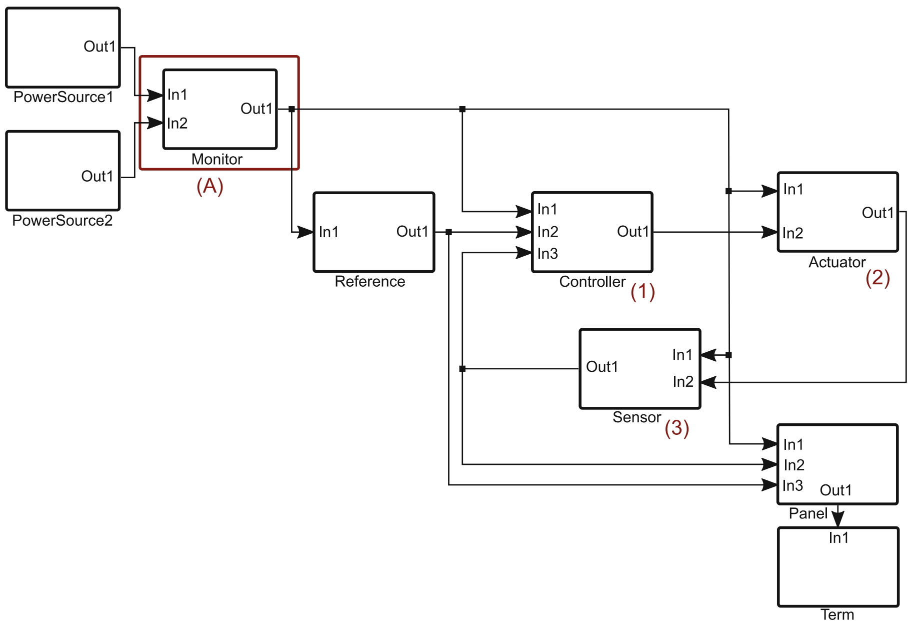


Fig. 4. Block diagram of the ACS provided by EMBRAER (nominal model)

Table 2. Annotations table of the ACS provided by EMBRAER

Component	Deviation	Port	Annotation
PowerSource	LowPower	Out1	PowerSourceFailure
Monitor	LowPower	Out1	(SwitchFailure AND (LowPower-In1 OR LowPower-In2)) OR (LowPower-In1 AND LowPower-In2)

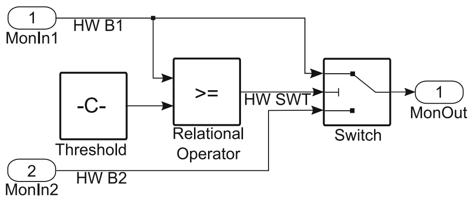


Fig. 5. Internal diagram of the monitor component (Fig. 4(A)).

same results as shown in [8]. To simplify formulas writing, we associate the fault events as:

$$\begin{aligned} B_1 &= \text{LowPower-In1} \\ B_2 &= \text{LowPower-In2} \\ F &= \text{SwitchFailure} \end{aligned}$$

The power source has only two possible operational modes: (i) the power source works as expected, providing a nominal value of $12V$, and (ii) is has an internal failure B_i, and its operational mode is "low power". In AL it is modelled as:

$$PowerSource_i = \{(B_i, LP), (\neg B_i, \text{Nominal}\, 12V)\} \tag{23}$$

where LP is the LowPower failure. The expression $PowerSource_i$ is healthy.

The monitor is a bit different because its behaviour depends not only on internal faults, but also on its inputs. We will now use the predicates notation defined in Sect. 3.4 to express fault propagation. As the monitor has two inputs and its behaviour is described in Fig. 5, then it is a function of the expressions of both inputs:

$$\begin{aligned} &Monitor_{bool}(in_1, in_2) = \\ &\qquad \text{modes}\,(in_1, \langle \text{out}\,(in_1) = \text{Nominal}\, X\rangle \wedge \neg F) \cup \\ &\qquad \text{modes}\,(in_2, \neg\langle \text{out}\,(in_1) = \text{Nominal}\, X\rangle \wedge \neg F) \cup \\ &\qquad \text{modes}\,(in_2, \langle \text{out}\,(in_1) = \text{Nominal}\, X\rangle \wedge F) \cup \\ &\qquad \text{modes}\,(in_1, \neg\langle \text{out}\,(in_1) = \text{Nominal}\, X\rangle \wedge F) \end{aligned} \tag{24}$$

where X is an unbound variable and assumes any value. The expression states the following:

– The monitor output is the same as in_1 if the output of in_1 *is* nominal and *there is no* internal failure in the monitor:

$$\text{modes}\,(in_1, \langle \text{out}\,(in_1) = \text{Nominal}\, X\rangle \wedge \neg F)$$

– The monitor output is the same as in_2 if the output of in_1 *is not* nominal and *there is no* internal failure in the monitor:

$$\text{modes}\,(in_2, \neg\langle \text{out}\,(in_1) = \text{Nominal}\, X\rangle \wedge \neg F)$$

– The monitor output is the converse of the previous two conditions if the internal failure F is active:

$$\begin{gathered} \text{modes}\,(in_2, \langle \text{out}\,(in_1) = \text{Nominal}\, X\rangle \wedge F) \cup \\ \text{modes}\,(in_1, \neg\langle \text{out}\,(in_1) = \text{Nominal}\, X\rangle \wedge F) \end{gathered}$$

The operational modes (observed behaviour) of the monitor depend on: (i) its internal fault, and (ii) propagated errors from its inputs. Composing the monitor with the two power sources, we obtain the AL expression of a power supply subsystem $System_{\text{bool}}$:

$$\begin{aligned}
&= Monitor_{bool}\left(PowerSource_1, PowerSource_2\right) \\
&= \text{modes}\left(in_1, \neg B_1 \wedge \neg F\right) \cup \text{modes}\left(in_2, \neg\neg B_1 \wedge \neg F\right) \cup \\
&\quad \text{modes}\left(in_2, \neg B_1 \wedge F\right) \cup \text{modes}\left(in_1, \neg\neg B_1 \wedge F\right) && \text{by Eq. (19)} \\
&= \text{modes}\left(in_1, \neg B_1 \wedge \neg F\right) \cup \text{modes}\left(in_2, B_1 \wedge \neg F\right) \cup \\
&\quad \text{modes}\left(in_2, \neg B_1 \wedge F\right) \cup \text{modes}\left(in_1, B_1 \wedge F\right) && \text{by simplification} \\
&= \left\{\left(P_i \wedge \neg B_1 \wedge \neg F, O_i\right) \mid \left(P_i, O_i\right) \in in_1\right\} \cup \\
&\quad \left\{\left(P_i \wedge B_1 \wedge \neg F, O_i\right) \mid \left(P_i, O_i\right) \in in_2\right\} \cup \\
&\quad \left\{\left(P_i \wedge \neg B_1 \wedge F, O_i\right) \mid \left(P_i, O_i\right) \in in_2\right\} \cup \\
&\quad \left\{\left(P_i \wedge B_1 \wedge F, O_i\right) \mid \left(P_i, O_i\right) \in in_1\right\} && \text{by Eq. (20)} \\
&= \left\{\left(B_1 \wedge \neg B_1 \wedge \neg F, LP\right),\right. \\
&\quad \left(\neg B_1 \wedge \neg B_1 \wedge \neg F, \text{Nominal } 12V\right), \\
&\quad \left(B_2 \wedge B_1 \wedge \neg F, LP\right), \\
&\quad \left(\neg B_2 \wedge B_1 \wedge \neg F, \text{Nominal } 12V\right), \\
&\quad \left(B_2 \wedge \neg B_1 \wedge F, LP\right), \\
&\quad \left(\neg B_2 \wedge \neg B_1 \wedge F, \text{Nominal } 12V\right), \\
&\quad \left(B_1 \wedge B_1 \wedge F, LP\right), \\
&\quad \left.\left(\neg B_1 \wedge B_1 \wedge F, \text{Nominal } 12V\right)\right\} && \text{replacing vars}
\end{aligned}$$

Simplifying and applying H_1, we obtain:

$$\begin{aligned}
&H_1\left(System_{\text{bool}}\right) = \\
&\qquad \left\{\left(\neg B_1 \wedge \neg F, \text{Nominal } 12V\right), \left(B_2 \wedge B_1 \wedge \neg F, LP\right),\right. \\
&\qquad \left(\neg B_2 \wedge B_1 \wedge \neg F, \text{Nominal } 12V\right), \left(B_2 \wedge \neg B_1 \wedge F, LP\right), \\
&\qquad \left.\left(\neg B_2 \wedge \neg B_1 \wedge F, \text{Nominal } 12V\right), \left(B_1 \wedge F, LP\right)\right\}
\end{aligned}$$

Applying, H_3, we simplify to:

$$\begin{aligned}
&H_3 \circ H_1\left(System_{\text{bool}}\right) \\
&\quad = \left\{ \left(\begin{array}{c} \left(\neg B_1 \wedge \neg F\right) \vee \\ \left(B_1 \wedge \neg B_2 \wedge \neg F\right) \vee, \text{Nominal } 12V \\ \left(\neg B_1 \wedge \neg B_2 \wedge F\right) \end{array} \right), \right. \\
&\qquad \left. \left(\begin{array}{c} \left(B_1 \wedge B_2 \wedge \neg F\right) \vee \\ \left(\neg B_1 \wedge B_2 \wedge F\right) \vee, LP \\ \left(B_1 \wedge F\right) \end{array} \right) \right\} \\
&\quad = \left\{\left(\left(\neg B_1 \wedge \neg B_2\right) \vee \neg F \wedge \left(\neg B_1 \vee \neg B_2\right), \text{Nominal } 12V\right),\right. \\
&\qquad \left.\left(F \wedge \left(B_1 \vee B_2\right) \vee \left(B_1 \wedge B_2\right), LP\right)\right\}
\end{aligned}$$

The monitor expression is H_2-healthy (the predicates are complete), thus:

$$H_2 \circ H_3 \circ H_1 \left(System_{\text{bool}} \right) = H_3 \circ H_1 \left(System_{\text{bool}} \right)$$

The resulting expression for the monitor after applying all healthiness conditions is:

$$\begin{aligned} \mathrm{H} \left(System_{\text{bool}} \right) = \{ & \left(\left(\neg B_1 \land \neg B_2 \right) \lor \neg F \land \left(\neg B_1 \lor \neg B_2 \right), \text{Nominal}\, 12V \right), \\ & \left(F \land \left(B_1 \lor B_2 \right) \lor \left(B_1 \land B_2 \right), LP \right) \} \end{aligned} \tag{25}$$

The operational modes of $System_{\text{bool}}$ is either Nominal $12V$ or LP (low power).

Finally, we obtain the *low power* structure expression (see Table 2) using the predicates notation:

$$\langle \text{out} \left(System_{\text{bool}} \right) = LP \rangle \iff F \land \left(B_1 \lor B_2 \right) \lor \left(B_1 \land B_2 \right)$$

The monitor expression also indicates that if the switch is operational ($\neg F$) and at least one PowerSource is operational ($\neg B_1 \lor \neg B_2$), the monitor output is nominal. But if at least one PowerSource is faulty ($B_1 \lor B_2$) and the monitor has an internal failure (F) the system is not operational. These two sentences written in AL using predicates notation are:

$$\begin{aligned} & \text{activation} \left(System_{\text{bool}}, \neg F \land \left(\neg B_1 \lor \neg B_2 \right) \right) \\ & \quad = \{ O | \left(P, O \right) \in \mathrm{H} \left(System_{\text{bool}} \right) \land \\ & \quad\quad \text{tautology} \left(\neg F \land \left(\neg B_1 \lor \neg B_2 \right) \implies P \right) \} \quad && [\text{by Eq. (21)}] \\ & \quad = \{ \text{Nominal}\, 12V \} && [\text{by simplification}] \quad (26a) \\ & \text{activation} \left(System_{\text{bool}}, F \land \left(B_1 \lor B_2 \right) \right) \\ & \quad = \{ O | \left(P, O \right) \in \mathrm{H} \left(System_{\text{bool}} \right) \land \\ & \quad\quad \text{tautology} \left(F \land \left(B_1 \lor B_2 \right) \implies P \right) \} && [\text{by Eq. (21)}] \\ & \quad = \{ LP \} && [\text{by simplification}] \quad (26b) \end{aligned}$$

Now, let's consider the same system but with a subtle modification. As shown in [8], the order of the occurrence of faults may be relevant, and the qualitative and quantitative analyses results may be different than those results without considering the order of the occurrence of faults. Observing Fig. 5, we see that if F activates before a failure in the first input of the monitor, then it would display a nominal behaviour, because the internal failure F anticipates switching to the second input. On the other hand, if the first input fails before F, then the monitor would switch to the second input, then switch back, due to the internal failure. We obtain the following expression for the monitor, now using the ATF:

$$\begin{aligned} & Monitor_{ATF} \left(in_1, in_2 \right) = \\ & \qquad \text{modes} \left(in_1, \langle \text{out} \left(in_1 \right) = \text{Nominal}\, X \rangle \land \neg F \right) \cup \\ & \qquad \text{modes} \left(in_2, \neg \langle \text{out} \left(in_1 \right) = \text{Nominal}\, X \rangle \land \neg F \right) \cup \\ & \qquad \text{modes} \left(in_2, \langle \text{out} \left(in_1 \right) = \text{Nominal}\, X \rangle \land F \right) \cup \\ & \qquad \text{modes} \left(in_1, \neg \langle \text{out} \left(in_1 \right) = \text{Nominal}\, X \rangle \rightarrow F \right) \cup \\ & \qquad \text{modes} \left(in_2, F \rightarrow \neg \langle \text{out} \left(in_1 \right) = \text{Nominal}\, X \rangle \right) \end{aligned} \tag{27}$$

where X is an unbound variable and assumes any value.

The difference to $System_{\text{bool}}$ (Eq. (24)) is only the finer analysis of the cases of erroneous behaviour of the first input and an internal failure. Note that the finer analysis splits the predicate

$$\neg \langle \text{out}\,(in_1) = \text{Nominal}\,12V\rangle \wedge F \qquad (\text{activates } in_1)$$

into:

$$\neg \langle \text{out}\,(in_1) = \text{Nominal}\,12V\rangle \rightarrow F \qquad (\text{activates } in_1)$$

and

$$F \rightarrow \neg \langle \text{out}\,(in_1) = \text{Nominal}\,12V\rangle \qquad (\text{activates } in_2)$$

We can assure that such a split is complete because the predicate notation evaluates to B_1. Thus the operands satisfy all temporal properties (Eqs. (8a) to (8d)) and events independence (Eq. (10)), thus the law shown in Eq. (12b) is valid. For case (i), the expected behaviour is the same as in_1 because the system switches to in_2, but then an internal failure occurs, and it switches back to in_1. For case (ii), it switches to in_2 due to an internal failure, then the first input fails, so the behaviour is similar to the nominal behaviour (see the second *modes* in Eq. (27)).

Following the similar expansions of Eq. (24), we obtain:

$$\begin{aligned} System_{ATF} =& Monitor_{ATF}\,(PowerSource_1, PowerSource_2)\\ =& \{(B_1 \wedge \neg B_1 \wedge \neg F, LP)\,,\\ & (\neg B_1 \wedge \neg B_1 \wedge \neg F, \text{Nominal}\,12V)\,,\\ & (B_2 \wedge B_1 \wedge \neg F, LP)\,,\\ & (\neg B_2 \wedge B_1 \wedge \neg F, \text{Nominal}\,12V)\,,\\ & (B_2 \wedge \neg B_1 \wedge F, LP)\,,\\ & (\neg B_2 \wedge \neg B_1 \wedge F, \text{Nominal}\,12V)\,,\\ & (B_1 \wedge B_1 \rightarrow F, LP)\,,\\ & (\neg B_1 \wedge B_1 \rightarrow F, \text{Nominal}\,12V)\}\,,\\ & (B_2 \wedge F \rightarrow B_1, LP)\,,\\ & (\neg B_2 \wedge F \rightarrow B_1, \text{Nominal}\,12V)\} \end{aligned}$$

Simplifying and applying H_1 to remove contradictions, we obtain:

$$\begin{aligned} &\text{H}_1\,(System_{ATF}) =\\ &\quad \{(\neg B_1 \wedge \neg F, \text{Nominal}\,12V)\,, (B_2 \wedge B_1 \wedge \neg F, LP)\,,\\ &\quad (\neg B_2 \wedge B_1 \wedge \neg F, \text{Nominal}\,12V)\,, (B_2 \wedge \neg B_1 \wedge F, LP)\,,\\ &\quad (\neg B_2 \wedge \neg B_1 \wedge F, \text{Nominal}\,12V)\,, (B_1 \rightarrow F, LP)\,,\\ &\quad (B_2 \wedge F \rightarrow B_1, LP)\,, (\neg B_2 \wedge F \rightarrow B_1, \text{Nominal}\,12V)\} \end{aligned}$$

Applying H_3 to remove redundant terms with identical operational modes and using the rules shown in Sect. 2.3, we simplify to:

$$
\begin{aligned}
&\mathrm{H}_3 \circ \mathrm{H}_1\left(System_{\mathrm{ATF}}\right) \\
&= \left\{ \left(\begin{matrix} (\neg B_1 \wedge \neg F) \vee \\ (B_1 \wedge \neg B_2 \wedge \neg F) \vee \\ (\neg B_1 \wedge \neg B_2 \wedge F) \vee \\ (\neg B_2 \wedge F \rightarrow B_1) \end{matrix}, \mathrm{Nominal}\, 12V \right), \right. \\
&\quad \left. \left(\begin{matrix} (B_1 \wedge B_2 \wedge \neg F) \vee \\ (\neg B_1 \wedge B_2 \wedge F) \vee \\ (B_1 \rightarrow F) \vee \\ (B_2 \wedge F \rightarrow B_1) \end{matrix}, LP \right) \right\} \\
&= \{((\neg B_1 \wedge \neg B_2) \vee \neg F \wedge (\neg B_1 \vee \neg B_2) \vee \\
&\quad \neg B_2 \wedge F \rightarrow B_1, \mathrm{Nominal}\, 12V), \\
&\quad ((B_1 \wedge B_2) \vee (\neg B_1 \wedge B_2 \wedge F) \vee (\neg B_2 \wedge B_1 \rightarrow F), LP)\}
\end{aligned}
$$

The monitor expression is H_2-healthy. Simplifying Boolean operators as usual, the XBefore expression:

$$\neg B_2 \wedge F \rightarrow B_1 \vee \neg B_2 \wedge B_1 \rightarrow F$$

simplifies to

$$\neg B_2 \wedge F \wedge B_1 \qquad \text{by Eq. (12b)}$$

Thus:

$$\mathrm{H}_2 \circ \mathrm{H}_3 \circ \mathrm{H}_1\left(System_{\mathrm{ATF}}\right) = \mathrm{H}_3 \circ \mathrm{H}_1\left(System_{\mathrm{ATF}}\right)$$

The resulting expression for the monitor after applying all healthiness conditions is:

$$
\begin{aligned}
\mathrm{H}\left(System_{ATF}\right) = \{&((\neg B_1 \wedge \neg B_2) \vee \neg F \wedge (\neg B_1 \vee \neg B_2) \vee \\
&\neg B_2 \wedge F \rightarrow B_1, \mathrm{Nominal}\, 12V), \\
&((B_1 \wedge B_2) \vee (\neg B_1 \wedge B_2 \wedge F) \vee \\
&(\neg B_2 \wedge B_1 \rightarrow F), LP)\}
\end{aligned} \tag{28}
$$

Finally, we obtain the *low power* structure expression of the monitor using the predicates notation:

$$\langle \mathrm{out}\left(System_{\mathrm{ATF}}\right) = LP \rangle \iff (B_1 \wedge B_2) \vee (\neg B_1 \wedge B_2 \wedge F) \vee (\neg B_2 \wedge B_1 \rightarrow F)$$

Thus, $System_{\mathrm{ATF}}$ fails with LP if:

– Both power sources fail;
– The monitor fails to detect the nominal state of the first power source and the second power source is in a failure state;

– The monitor fails to detect the failure state of the first power source (the monitor fails after the failure of the first power source).

Note that if the monitor fails before the failure of the first power source, it fails to detect the operational mode of the first power source and switches to the second power source, which is in a nominal state (see expression $\neg B_2 \wedge F \rightarrow B_1$ in Eq. (28)).

5 Conclusion

In this work we proposed a parametrized logic that enables the analysis of systems depending on the expressiveness of a given algebra and a given set of operational modes. If ATF is used as a parameter, then the order of occurrence of faults can be considered. Although the logic is not as detailed as AADL, the predicates notation in conjunction with the ATF provides a richer assertion framework. Also, it is possible to verify non-determinism on the model, by: (i) verifying its existence with the nondeterministic function, (ii) providing an expression and obtaining the possible operational modes with the activation function, or (iii) using the predicates notation to obtain a predicate that enables two or more operational modes.

The AADL is extensible. The work reported in [2] shows an extension to perform dependability analysis through state machines and expressions on fault events and operational modes. Although such an extension captures system behaviour, operational mode activation conditions are expressed in state transitions in combination with an extension of Boolean expressions (not related to order). Our work relates operational modes and fault occurrences order explicitly.

As presented in [8], TFTs and DFTs structure expressions can be written as formulas in ATF. As the root events of TFTs and DFTs represent operational modes of a system, the ATF can be used to associate root events with operational modes, thus allowing the combination of two or more fault trees.

Although the properties of AL require that the inner algebra provides tautology and contradiction, and we used ATF in the case study, we did not show tautology and contradiction for ATF. Instead, we used a law to reduce the ATF expression to a Boolean expression. The methodology to check tautology and contradiction in ATF is a future work.

The original expression shown in the case study was already H_2-healthy. The second healthiness condition about completeness uses the concept of undefined value to make any expression H_2-healthy. Algebraically it is fine, but in practice, the property should be met originally, thus the initial expression is already H_2-healthy. This property should be used as an alert to the analyst if it not met originally.

Acknowledgements. We would like to thank **Alexander Romanovsky**, **Zoe Andrews** and **Richard Payne** for all discussions about fault modelling and dependability. This work was funded by CNPq, grants 476821/2011-8, 442859/2014-7, and 246956/2012-7, and by FACEPE grant IBPG-0408-1.03/11. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES, http://www.ines.org.br), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

References

1. SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996
2. SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: ARINC653 Annex, Annex C: Code Generation Annex, Annex E: Error Model Annex. Technical report, SAE International (2015)
3. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **C–27**(6), 509–516 (1978)
4. ANAC. Aeronautical Product Certification. DOU No. 230, Seção 1, p. 28, 01 December 2011, (2011)
5. Andrews, J.D.: The use of not logic in fault tree analysis. Qual. Reliab. Eng. Int. **17**(3), 143–150 (2001)
6. Boute, R.T.: The binary decision machine as programmable controller. Euromicro Newslett. **2**(1), 16–22 (1976)
7. Didier, A.L.R., Mota, A.: Identifying hardware failures systematically. In: Gheyi, R., Naumann, D. (eds.) Formal Methods: Foundations and Applications. Lecture Notes in Computer Science, vol. 7498, pp. 115–130. Springer, Heidelberg (2012)
8. Didier, A.L.R., Mota, A.: An algebra of temporal faults. Inf. Syst. Front. **18**, 967–980 (2016)
9. Dugan, J.B., Bavuso, S.J., Boyd, M.A.: Dynamic fault-tree models for fault-tolerant computer systems. IEEE Trans. Reliab. **41**(3), 363–377 (1992)
10. FAA. RTCA, Inc., Document RTCA/DO-178B. U.S. Dept. of Transportation, Federal Aviation Administration, Washington, D.C. (1993)
11. FAA. Part 25 - Airworthiness Standards: Transport Category Airplanes. report, Federal Aviation Administration (FAA), USA (2007)
12. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis & Design Language (AADL): An Introduction. CMU/SEI–2006–TN–011, February 2006
13. Givant, S., Halmos, P.: Introduction to Boolean Algebras. Undergraduate Texts in Mathematics, vol. XIV. Springer, New York (2009)
14. Hoare, C.A.R., He, J.: Unifying Theories of Programming, vol. 14. Prentice Hall, Englewood Cliffs (1998)
15. Koren, I., Krishna, C.M.: Fault Tolerant Systems. Morgan Kaufmann Publishers Inc., San Francisco (2007)
16. Merle, G.: Algebraic modelling of Dynamic Fault Trees, contribution to qualitative and quantitative analysis. Theses, École normale supérieure de Cachan - ENS Cachan (2010)
17. Merle, G., Roussel, J.-M., Lesage, J.-J.: Algebraic determination of the structure function of Dynamic Fault Trees. Reliab. Eng. Syst. Saf. **96**(2), 267–277 (2011)
18. Merle, G., Roussel, J.-M., Lesage, J.-J.: Dynamic fault tree analysis based on the structure function. In: 2011 Proceedings - Annual Reliability and Maintainability Symposium, January 2011

19. Merle, G., Roussel, J.-M., Lesage, J.-J.: Quantitative analysis of dynamic fault trees based on the structure function. Qual. Reliab. Eng. Int. **30**(1), 143–156 (2014)
20. Merle, G., Roussel, J.-M., Lesage, J.-J., Bobbio, A.: Probabilistic algebraic analysis of fault trees with priority dynamic gates and repeated events. IEEE Trans. Reliab. **59**(1), 250–261 (2010)
21. O'Connor, P.D.T., Newton, D., Bromley, R.: Practical Reliability Engineering. Wiley, Hoboken (2002)
22. Oliva, S.: Non-coherent fault trees can be misleading. e-J. Syst. Saf. **42**(3), 1–5 (2006)
23. Tannous, O., Xing, L., Dugan, J.B.: Reliability analysis of warm standby systems using sequential BDD. In: 2011 Proceedings - Annual Reliability and Maintainability Symposium, January 2011
24. Vesely, W., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault Tree Handbook. Number NUREG-0492. US Independent Agencies and Commissions (1981)
25. Walker, M.D.: Pandora: a logic for the qualitative analysis of temporal fault trees. Ph.D. thesis, University of Hull (2009)
26. Walker, M.D., Papadopoulos, Y.: Synthesis and analysis of temporal fault trees with PANDORA: the time of Priority AND gates. Nonlinear Anal. Hybrid Syst. **2**(2), 368–382 (2008)
27. Walker, M.D., Papadopoulos, Y.: Qualitative temporal analysis: towards a full implementation of the fault tree handbook. Control Eng. Pract. **17**(10), 1115–1125 (2009)
28. Walker, M.D., Papadopoulos, Y.: A hierarchical method for the reduction of temporal expressions in Pandora. In: Proceedings of the First Workshop on DYnamic Aspects in DEpendability Models for Fault-Tolerant Systems, DYADEM-FTS 2010, pp. 7–12. ACM, New York (2010)
29. Xing, L., Tannous, O., Dugan, J.B.: Reliability analysis of nonrepairable cold-standby systems using sequential binary decision diagrams. IEEE Trans. Syst. Man Cybern. A **42**(3), 715–726 (2012)

Author Index

S.H. Rubin and T. Bouabana-Tebibel (eds.), *Quality Software Through Reuse and Integration*, Advances in Intelligent Systems and Computing 561,
DOI 10.1007/978-3-319-56157-8