Arend Rensink
Jesús Sánchez Cuadrado (Eds.)

LNCS 10888

Theory and Practice of Model Transformation

11th International Conference, ICMT 2018
Held as Part of STAF 2018
Toulouse, France, June 25–26, 2018, Proceedings

Springer

Arend Rensink · Jesús Sánchez Cuadrado (Eds.)

Theory and Practice of Model Transformation

11th International Conference, ICMT 2018
Held as Part of STAF 2018
Toulouse, France, June 25–26, 2018
Proceedings

Springer

Editors
Arend Rensink
University of Twente
Enschede
The Netherlands

Jesús Sánchez Cuadrado
University of Murcia
Murcia
Spain

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-93316-0 ISBN 978-3-319-93317-7 (eBook)
https://doi.org/10.1007/978-3-319-93317-7

LNCS Sublibrary: SL2 – Programming and Software Engineering

Printed on acid-free paper

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

Foreword

Software Technologies: Applications and Foundations (STAF) is a federation of leading conferences on software technologies. It provides a loose umbrella organization with a Steering Committee that ensures continuity. The STAF federated event takes place annually. The participating conferences and workshops may vary from year to year, but they all focus on foundational and practical advances in software technology. The conferences address all aspects of software technology, from object-oriented design, testing, mathematical approaches to modeling and verification, transformation, model-driven engineering, aspect-oriented techniques, and tools. STAF was created in 2013 as a follow-up to the TOOLS conference series that played a key role in the deployment of object-oriented technologies. TOOLS was created in 1988 by Jean Bézivin and Bertrand Meyer and STAF 2018 can be considered its 30th birthday.

STAF 2018 took place in Toulouse, France, during June 25–29, 2018, and hosted: five conferences, ECMFA 2018, ICGT 2018, ICMT 2018, SEFM 2018, TAP 2018, and the Transformation Tool Contest TTC 2018; eight workshops and associated events. STAF 2018 featured seven internationally renowned keynote speakers, welcomed participants from all around the world, and had the pleasure to host a talk by the founders of the TOOLS conference Jean Bézivin and Bertrand Meyer.

The STAF 2018 Organizing Committee would like to thank (a) all participants for submitting to and attending the event, (b) the Program Committees and Steering Committees of all the individual conferences and satellite events for their hard work, (c) the keynote speakers for their thoughtful, insightful, and inspiring talks, and (d) the Ecole Nationale Supérieure d'Electrotechnique, Electronique, Hydraulique et Télécommunications (ENSEEIHT), the Institut National Polytechnique de Toulouse (Toulouse INP), the Institut de Recherche en Informatique de Toulouse (IRIT) for hosting us and for their support. A special thanks goes to all the members of the Software and System Reliability Department of the IRIT laboratory and the members of the INP-Act SAIC, coping with all the foreseen and unforeseen work to prepare a memorable event.

June 2018

Marc Pantel
Jean-Michel Bruel

Preface

This volume contains the papers presented at ICMT 2018: the 11th International Conference on Model Transformation held during June 25–26, 2018, in Toulouse, France, as part of the STAF 2018 (Software Technologies: Applications and Foundations) conference series. ICMT is the premier forum for researchers and practitioners from all areas of model transformation.

Modeling is a key element in reducing the complexity of software systems during their development and maintenance. Model transformations are essential for elevating models from documentation elements to first-class artifacts. Transformations also play a key role in analyzing models to reveal conceptual flaws or highlight quality bottlenecks and in integrating heterogeneous tools into unified tool chains.

Model transformation encompasses a variety of technical spaces, including modelware, grammarware, dataware, and ontoware, a variety of model representations, e.g., trees vs. graphs, and a variety of transformation paradigms including rule-based transformations, term re-writing, and manipulations of objects in general-purpose programming languages. Moreover, in other fields such as compiler construction, the use of transformations is likewise essential. Identifying means to reuse and share knowledge between fields is also of interest.

The study of model transformation includes foundations, structuring mechanisms, and properties, such as modularity and composability, transformation languages, techniques, and tools. An important goal of the field is the development of high-level model transformation languages, providing transformations that are amenable to higher-order model transformations or tailored to specific transformation problems. At the same time, usable and scalable verification techniques for model transformations are essential for the practical development of the field. Another key challenge is the efficient execution of model queries and transformations by scalable transformation engines. Novel algorithms as well as innovative (e.g., distributed) execution strategies and domain-specific optimizations are sought in this respect. Model transformations have become artifacts that need to be managed in a structured way, resulting in developing methodology and tools to deal with versioning, (co-)evolution, reuse, etc. Correctness of model transformations has to be guaranteed as well.

This year ICMT 2018 received 24 submissions. Each submission was reviewed by three Program Committee members. After an online discussion period, the Program Committee accepted nine papers as part of the conference program. These papers included research, application, and tool demonstration papers presented in the context of four sessions on verification of model transformations, model transformation tools, transformation reuse, and graph transformations. In addition, we had an invited paper by our keynote speaker, Markus Voelter, about the design and evolution of KernelF.

A lot of people contributed to the success of ICMT 2018. We are grateful to the Program Committee members and reviewers for the timely delivery of thorough reviews and constructive discussions under a very tight review schedule. We also thank

Markus Voelter for his excellent keynote talk. Last but not least, we would like to thank the authors, who constitute the heart of the model transformation community, for their enthusiasm and hard work.

The organization of STAF made for a successful conference. We thank the local organizers, and in particular the general chairs, Marc Pantel and Jean-Michel Bruel; and we thank the Ecole Nationale Supérieure d'Electrotechnique, Electronique, Hydraulique et Télécommunications (ENSEEIHT), the Institut National Polytechnique de Toulouse (Toulouse INP), and the Institut de Recherche en Informatique de Toulouse (IRIT) for hosting us and for their support.

June 2018

Arend Rensink
Jesús Sánchez Cuadrado

Organization

Program Committee

Anthony Anjorin	Paderborn University, Germany
Rubby Casallas	University of Los Andes, Colombia
Marsha Chechik	University of Toronto, Canada
Antonio Cicchetti	Mälardalen University, Sweden
Benoit Combemale	IRIT, University of Toulouse, France
Davide Di Ruscio	Università degli Studi dell'Aquila, Italy
Juergen Dingel	Queen's University, Canada
Gregor Engels	University of Paderborn, Germany
Martin Gogolla	Database Systems Group, University of Bremen, Germany
Esther Guerra	Universidad Autónoma de Madrid, Spain
Soichiro Hidaka	Hosei University, Japan
Ludovico Iovino	Gran Sasso Science Institute, Italy
Frédéric Jouault	TRAME Team, ESEO, France
Timo Kehrer	Humboldt-Universität zu Berlin, Germany
Dimitris Kolovos	University of York, UK
Leen Lambers	Hasso-Plattner-Institut, Universität Potsdam, Germany
Yngve Lamo	Western Norway University of Applied Sciences, Norway
Tanja Mayerhofer	Vienna University of Technology, Austria
Richard Paige	University of York, UK
Bernhard Rumpe	RWTH Aachen University, Germany
Houari Sahraoui	University of Montreal, Canada
Andy Schürr	TU Darmstadt, Germany
Eugene Syriani	University of Montreal, Canada
Gabriele Taentzer	Philipps-Universität Marburg, Germany
Massimo Tisi	Inria, France
Mark Van Den Brand	Eindhoven University of Technology, The Netherlands
Hans Vangheluwe	University of Antwerp, Belgium, and McGill University, Canada
Daniel Varro	Budapest University of Technology and Economics, Hungary
Edward Willink	Willink Transformations Ltd., UK
Manuel Wimmer	Business Informatics Group, Vienna University of Technology, Austria
Vadim Zaytsev	University of Amsterdam, The Netherlands
Steffen Zschaler	King's College London, UK

Additional Reviewers

Bertram, Vincent
Bousse, Erwan
Burgueno, Loli
Cleophas, Loek
Eikermann, Robert
Kuhlmann, Mirco
Kästner, Andreas
Leblebici, Erhan
Michael, Judith
Neubauer, Patrick
Rabbi, Fazle
Sohr, Karsten
Zolotas, Athanasios

Contents

Invited Paper

The Design, Evolution, and Use of KernelF
An Extensible and Embeddable Functional Language

Markus Voelter(✉)

Stuttgart, Germany
http://voelter.de

Abstract. KernelF is a functional language built on top of MPS. It is designed to be highly extensible and embeddable in order to support its use at the core of domain-specific languages, realising an approach we sometimes call Funclerative Programming. "Funclerative" is of course a mash-up of "functional" and "declarative" and refers to the idea of using functional programming in the small, and declarative language constructs for the larger-scale, often domain-specific, structures in a program. We have used KernelF in a wide range of languages including health and medicine, insurance contract definition, security analysis, salary calculations, smart contracts and language-definition. In this paper, I illustrate the evolution of KernelF over the last two years. I discuss requirements on the language, and how those drove design decisions. I showcase a couple of the DSLs we built on top of KernelF to explain how MPS was used to enable the necessary language modularity. I demonstrate how we have integrated the Z3 solver to verify some aspects of programs. I present the architecture we have used to use KernelF-based DSLs in safety-critical environments. I close the keynote with an outlook on how KernelF might evolve in the future, and point out a few challenges for which we don't yet have good solutions.

Keywords: Domain-specific languages · Language modularity
Functional Language · Language engineering · Meta programming

1 Introduction

1.1 Funclerative Programming

Functional programming is suitable for programming in the small [6], for compact algorithms. It is not ideally suited for programming in the large. Reasons include the lack of means for grouping functions into interfaces, hiding information, and defining contracts. To compensate for this, languages combine the functional paradigm with other paradigms, such as object-oriented programming in Scala [20]. Higher-level frameworks such as MapReduce [5] also provide more

M. Voelter—Independent/itemis.

coarse-grained control over program execution that goes beyond the typical building blocks of functional languages: function calls, higher order functions and monads.

KernelF combines functional programming in the small, and declarative structures and behaviours in the large, in an approach we sometimes call "funclerative programming". Instead of using one particular paradigm for providing coarse-grained behaviors and structure to programs, we extend a functional core language with custom, domain-specific abstractions.

1.2 Domain-Specific Languages

The need for custom abstractions on top of a functional core arises from domain-specific languages (DSLs). In our industry work, we[1] develop many different DSLs in a wide variety of different domains (we show a few examples in Sect. 5). All except the very trivial ones require a "calculation core": arithmetics, comparison, logical expressions, as well as functions, records and enums. Functional programming is perfectly suited for this task, because the lack of side-effects makes programs easy to analyse, and hence, safe to integrate into a DSL.

However, for most real-world DSL, functional abstractions alone are not sufficient. Instead, higher-level abstractions for the coarse-grained, often stateful behaviors are required, such as state machines, data flow or imperative programming. Finally, these DSLs operate on domain-specific data structures such as treatment logs in healthcare, insurance products or contract definitions in logistics. Constructing these from functional abstractions (or classes/objects) alone is not practical, since the result would be too limiting in terms of notation, static analyzability and IDE support. Thus, a three layer architecture for DSLs is typical in our work:

- Layer 1: Functional abstractions
- Layer 2: Higher-level behaviors, based on established paradigms
- Layer 3: Domain-specific data structures

1.3 A Reusable Functional Kernel Language

The domain specificity resides mostly in layers two and three, so there is potential for reuse of the functional abstractions of layer one. KernelF, the language discussed in this paper, is a functional language optimized for reuse as layer one. To make this feasible, it must be extensible, restrictable and configurable.

Extension. Extension refers to adding additional language constructs to the language. For example, if KernelF is used to express guard conditions in the transitions of state machines, it must be possible to add new expressions that refer to the parameters of the events that trigger the transition. This must be possible *without* invasively modifying the definition of KernelF itself, and the

[1] 'We' refers to the team of languages engineers at itemis Stuttgart.

extension must comprise structure, notation, scoping and type systems. To further enhance the potential for reuse, independently developed extensions should be combinable, again without invasive modification of the definition of any of the used languages (a feature called extension composition in [9]).

Restriction. This refers to the ability to not expose certain language concepts to the end user; for example, a DSL might not need support for `enums` or option types, so it must be possible to remove all traces of those concepts from KernelF when it is used in a particular DSL. In particular, the associated keywords should not be recognised and the IDE should not propose all related concepts in the code completion menu.

Configurability. In KernelF, this refers specifically to the ability to replace the primitive types. Often, a DSL will come with its own notion of numbers or strings, and those must then be used by KernelF. This is not exactly the same problem as restriction or extension because the type system will internally rely on those primitive types. Consider a `size` operation; the type system must type this operation to whatever (positive) integral type used by the surrounding DSL, so the *primitive types used by built-in operators* must be configurable.

1.4 Design Guidelines for the Use in DSLs

KernelF is intended to be used as the calculation core of DSLs. Many of the users of these DSLs may not be programmers – most will certainly not be experts in functional programming. To make the language suitable for this purpose, it should adhere to the following guidelines, in addition to being extensible, restrictable and configurable, as discussed above.

Simplicity. Users should not be surprised or overwhelmed. Thus, the language should use familiar or easy to learn abstractions and notations wherever possible. Advanced concepts, such as function composition or monads are not suitable. More generally, the ability to allow users to define their own (structural or behavioral) abstractions in their programs can be limited (in the service of the goal of simplicity), because those can be provided in domain-specific language extensions. A subrequirement of simplicity is **readability**; it is particularly relevant because many of the potential users who write KernelF code will start out by *reading* KernelF code when reviewing code written by other users. Scaring prospective users away during the reading phase is not helpful.

Robustness. Since the users of the DSLs that embeds KernelF may not be experienced programmers the language should not have features that make it easy to make dangerous mistakes (such as pointer arithmetics, unbounded strings or overflow for numbers). To the contrary, the language should make "doing the right thing" easy. For example, handling errors should be integrated into the type system as opposed to C's approach of making checking of `errno` completely optional. It should also enable advanced analyses, for example, to detect unhandled cases in `switch`-style constructs.

IDE Integration. DSLs must come with good an IDE, otherwise they are not accepted by users. This means that the language should be designed so that it can be supported well by IDEs. Such support includes code completion, type checking, refactoring and debugging. IDE support is a way of achieving **writability**, i.e., the ease with which code can be written. Writability is often at odds with readability, which is why we optimize the syntax, once written, for readability, and use IDE support to simplify writing code. In addition, programs should be executable with a short turnaround, to support end users to "play" with the programs. Seeing what a program does is often easier for inexperienced users than imagining a program's behavior based on the program code.

Portability. The various languages into which KernelF is embedded use different means of execution such as code generation to Java and C, direct execution by interpreting the AST as well as transformation into intermediate languages for execution in cloud or mobile applications. KernelF should not contain features that prevent execution on any of these platforms. Also, while not a core feature of the language, a sufficient set of language tests should be provided to align the semantics of the various execution platforms.

1.5 Language Engineering and MPS

KernelF, and all the DSLs discussed in this paper, are built with Jetbrains MPS.[2] MPS is a language workbench [10], a tool for developing ecosystems of languages. MPS has been used for many interesting and significantly-sized languages over the last years, the biggest one probably being mbeddr [27,30], a set of C extensions optimized for embedded programming. MPS supports a wide range of modular language composition, in particular, extension and restriction are supported directly [24]. This is possible because of two fundamental properties of MPS. First, it relies on a projectional editor. Because projectional editors do not use parsing, no syntactic ambiguities arise when independently developed languages are combined. Second, MPS has been designed to not just develop one language, but ecosystems of collaborating languages. The formalisms for defining structure, type systems and scopes have all been designed with modularity and composition in mind; some details on language development with MPS as well as the general MPS language design philosophy is explained in [26]. We analyze MPS' suitability for modular language composition based on experience with mbeddr in [28] (the paper also evaluates MPS more generally). MPS' projectional editor also allows the use of a wide range of different notations such as tables, diagrams, math symbols as well as structured ("code") and unstructured ("prose") text [29], a feature we exploit extensively in the construction of DSLs. Projectional editors have historically had a bad reputation regarding usability. However, recent advances as implemented natively in MPS and in an extension called grammar cells [31] lead to good editor productivity and user acceptance [2].

[2] https://www.jetbrains.com/mps/.

2 KernelF Overview

2.1 Language

In this Sect. 1 point out the most important language features of KernelF. For all of them, [25] provides more details and code examples; for many of these features we also show examples part of the case studies in Sect. 5.

Purity and Effects. At its core, KernelF is a pure language. All expressions are effect-free. There are no variables, only named (local and global) values. All values, including collections are immutable. Of course, no sensible program can be written this way; but it is expected that the hosting DSL has domain-specific means of dealing with state. The core language thus supports effect tracking; each expression can describe whether it performs a `read` or `modify` effect.

Types, Literals and Operators. KernelF comes with Boolean and string types which work as one would expect. Numeric types comprise `int` and `real`, even though they are constrained out of the language in most of the DSLs. Instead, the `number[min|max]{decimals}` type is used, where the range and precision are explicitly specified. The type system performs range calculations for added type safety, and a change of the number of decimals has to be performed explicitly. The usual operators are defined on those types. No `null` values are supported, instead, the language supports option types (written as `opt<T>` for any type `T`). Type checking is static, and most types can be inferred (exceptions are function arguments, record members and return types for recursive functions). Finally, KernelF supports type definitions written as `type <name>: <OriginalType>`, useful for numbers with ranges/precisions, collections, and constraints (see below).

Loops and Conditionals. KernelF has no loops (except higher-order functions on collections). The basic $if < cond > then < expr - 1 > else < expr - 2 >$ distinguishes between two cases, whereas $alt| < cond - 1 > \Rightarrow < expr - 1 > ... < cond - n > \Rightarrow < expr - n > |$, laid out vertically, evaluates to `expr-i` if `cond-i` holds. `if` is also used to test options: `if isSome(v) then v else w` returns a `T` if `v` is of type `opt<T>` and `v` actually contains a value; it returns `w` if `v` contains a `none`. Various additional conditionals, in particular, decision tables and decision trees, are supported as part of a language extension.

Functions and Blocks. Functions use the usual syntax. Argument types have to be specified, the return type can be inferred except for recursive functions. The block expression, which is used instead of `let`, is written as `{<expr-1> ... <expr-n> <expr-ret>}`, laid out vertically. The block evaluates to `<expr-ret>`, and all other expressions must either have an effect or must be local values that are referenced downstream, written as `val v = <expr>`. Function types are written as `(T-1, T-2, ... T-n => T)`. Values of function types can executed using the `()` operator. Currying is supported via `f.bind(v)` if `f` is a function value. Lambdas are written as `|a-1: T-1, ... a-n: T-n: <expr>|` or, for lambdas with one argument which is then named `it`, as `|... it ...|`.

References to functions (which can be used as values for function types) are written as `:f` for any function `f`. KernelF also supports `extension` functions where the first argument can be written as the left side of a dot expression.

Error Handling. Language support for error handling relies on attempt types. Typically used with functions, if the function returns a `T` plus one of several errors, then the return type is `attempt<T|E-1,... E-n>` where the `E`s are error literals. Error values can be returned using `error(E)`; clients can react to errors using `try <expr> => <success> error <E-1> => <expr-1> ... error <E-n> => <expr-n>`, where `<expr>` has an attempt type, and the overall `try` evaluates to `<success>` if `<expr>` does not represent an error, or one of the `<expr-i>` if `expr` evaluates to an error literal `E-i`.

Collections. Lists, sets and maps are supported, together with the usual higher-order functions. Collections specify their element type, plus an optional size constraint, e.g. `list<T>[min|max]`. Literals use the same keyword; for example, `set(1, 2, 3)` or `map("Joe" => 12, "Jim" => 100)`.

User-Defined Types. KernelF supports `enum`s, both plain and with associated values. Tuples are supported as well, their types are written as `[<T-1>, ... <T-n>]` and their values are written as `[<expr-1>, ... <expr-n>]`. Member are accessed positionally, using array-access notation (`tuplevalue[p]`). Records are declared using a Pascal-like notation, record values are constructed via `#T(<expr-1>, ..., <expr-n>)` or a semi-graphical `build<T>` expression. Members are accessed using dot notation.

Constraints. KernelF supports constraints that are checked at runtime. They appear in several places, usually after the `where` keyword. `type` definitions can constrain the values; `records` can constrain their members, function can define pre- and postconditions, which typically constrain parameters or return values.

Boxes and Transactions. KernelF makes the notion of mutable state explicit through boxes. A value `v` of type `box<T>` represents an immutable reference to a mutable "memory location", of type `T` (similar to `refs` in Clojure [14]). The box contents can change over time, but each value in the box is immutable. `v.val` accesses the value inside the box, `v.update(<expr>)` sets the contents of the box to `expr`. Inside the `update`, the `it` expression represents the current value; this way, evolutions of the box contents can be written in a compact form, as in this example for a box `lb` of type `box<list<string>>`, where an additional value is appended to the contents of the box: `lb.update(it.plus("additionalEntry"))`. To make working with boxes safe, `.val` has a `read` effect, and `update` has a `modify` effect. Modifications to multiple boxes can be grouped into transactions. An failed update to any box, for example, because of a violation of a type constraint, rolls back the updates on all boxes.

State Machines. Once we had boxes to store evolving state, it was obvious that we need first-class support for expressing behavior that depends on state, i.e., state machines. KernelF state machines declare states, one of them initial, and the states can also be nested. Machines also declare events, which can optionally have arguments. State machines are passive, i.e., they have to be actively triggered by passing an event (and optionally, arguments) into an instance. A state owns transitions which, reacting to an event, bring the machine into a new target state. There are also automatic transitions that can be triggered by timeouts or other implicitly occurring events. State machines support entry and exit actions on states as well as transition actions.

2.2 Definition of the Semantics

The semantics of KernelF are given by the interpreter that ships with the language, together with a sufficiently large amount of test cases. No other formal definition of the language semantics is provided. To align the semantics of generators with the reference semantics given by the interpreter, one can simply generate the test cases to the target platform and then run them there – if all pass, the (relevant, functional) semantics are identical.

2.3 Tooling

Similar to the previous subsection, this one provides an overview over the tooling provided for KernelF; details are in [25]. Tooling is crucial for the acceptance of DSLs with their users, and all tooling discussed here for the core of KernelF is also available for the DSLs built on top of KernelF.

An **IDE**, implicitly provided by MPS, supports the usual editor features (syntax coloring, formatting, error markup, code completion, go to definition, find usages, tooltips) as well as version control integration including diff/merge support for arbitrary syntax. An **interpreter** is integrated directly into the IDE, supporting live execution of (suitably structured) programs. The interpreter is implemented in Java. A **code generator** to Java is available because most of the DSLs we build are ultimately mapped to Java code. To make semantic alignment with the interpreter easier, the generated code relies on the same persistent collections library as the interpreter, and also uses Java's `BigInteger`/`BigDecimal` for numbers. A **read-eval-print-loop** (REPL) is available for interactive use of the language. A **debugger** is available, it relies on rendering the execution trace as a tree, and overlaid directly over the code. One language module of KernelF supports writing **tests**, and, relying on the interpreter, they can be executed directly in the IDE, leading to the usual red/green visual feedback, directly in the code. Taken together, the REPL, tests, interpreter and debugger lead to a very "live" programming experience with quick feedback. To ensure test quality, KernelF supports **coverage measurement**, both structural (are all language features used, and how) and relative to the interpreter (are all parts of the interpreter executed). KernelF's test infrastructure also supports **test case generation** for language constructs that take arguments lists (functions, records) as well as **mutation testing** with interactive visualisation of the mutated code. Finally, we are in the process of integrating KernelF with the Z3 **solver** to provide advanced error checking.

3 Design Decision

Based on the goals for KernelF outlined in Sect. 1, we have made the design decisions outlined in this section.

3.1 General Design Decisions

Static Types. KernelF is statically typed. This means that every type is known by the IDE (as well as the interpreter or generator). If a user is interested in the type of an expression, they can always press `Ctrl-Shift-T` to see that type. This helps with the design goals of [SIMPLICITY] and [IDESUPPORT], but also with [ROBUSTNESS], because more aspects of the semantics can be checked statically in the IDE. For example, the number ranges discussed below are an example of such advanced checks.

Numeric Types. Instead of `int` and `real` types known from programming languages, KernelF uses the `number[min|max]{prec}` type. This is motivated primarily be [ROBUSTNESS] because it supports more end-user relevant checks. The type system performs simple range computations, such as those listed below.

– Number literals have a type that has a singleton range based on their value and number of decimal digits (e.g., `42.2` has the type `number[42.2|42.2]{1}`.
– Supertypes of numeric types merge the ranges (for example, the supertype of `number[5|5]`, `number[10|20]` and `number[30|50]` is `number[5|50]`. This is an over approximation (i.e., simplification in the type system implementation), because the type system could know that, for example, the value `25` is not allowed. However, to implement this, a number type would have to have *several* ranges; we decided that this would be too complicated (both for users and the language implementor) and induce performance penalties in type checking; so we decided to live with the over approximation.
– For arithmetic operations (currently `+, -, *` and `/`), the type system computes the correct result ranges; for example, if variables of type `number[0|5]` and `number[3|8]` are added, the resulting type is `number[3|13]`.
– A division *always* results in an infinite precision value; if a different precision is required, the `prevision<>()` operator has to be used.

We are making the simplifying tradeoffs consciously, because, in the extreme, we would have to implement a type system that supports dependent types (or abstract interpretation of code); this is clearly out of scope.

Type Inference. To avoid the need to explicitly specify types (especially the `attempt` types, collections and number types can get long), KernelF supports type inference; this supports both [READABILITY] and [WRITEABILITY]. The types of all constructs are inferred, with the following exceptions:

– Arguments and record members always require explicit types because they are declarations without associated expressions from which to infer the type.

– Recursive functions require a type because our type system cannot figure out the type of the body if this body contains a call to the same function.

If a required type is missing, an error message is annotated. Users can also use an intention on nodes that have optional type declarations (functions, constants) and have the IDE annotate the inferred type.

No Generics. KernelF does not support generics in user-defined functions, another consequence of our goal of [SIMPLICITY]. However, the built-in collections are generic (users explicitly specify the element type) and operations like `map`, `select`, or `tail` retain the type information thanks to the type system implementation in MPS. Domain-specific extensions can also define their own "generic" language extensions, similar to collections.

Option and Attempt Types. To support our goal of [ROBUSTNESS], the type system supports option types and attempt types. Options force client code to deal with the possibility of `null` (or `none`) values in programs. Similarly, attempt types deal systematically with errors and force the client code to handle them (or return the `attempt` type to its own caller).

No Exceptions. KernelF does not support exceptions. The reason is that these are hard or expensive to implement on some of the expected target platforms (such as generation to C); [PORTABILITY] would be compromised. Instead, attempt types and the constraints can be used for error handling.

No Reflection or Meta Programming. By deciding to rely on the language engineering capabilities of MPS, the language does not require an elaborate reflective type system (like Scala) or meta programming support to enable extension and embedding.

No Function Composition and Monads. We decided not to implement full support for monads; for our current use cases, this is acceptable and keeps the implementation of the type system simpler, which supports our goal of extensibility. Note that, because many operations and operators for `T` also work for `opt<T>`, users can defer dealing with options and errors until it makes sense to them; no nested `if isSome(...) ...` are required.

Effect Tracking and Types. Effect tracking is not implemented with the type system: an effect is not declared as part of the type signature of a function (or other construct). There are two reasons for this decision. First, for various technical reasons of the way the MPS type system engine works, this would be inefficient. Second, language extenders and embedders would have to deal with the resulting complexity when integrating with KernelF's type system. Instead, the analysis is based on the AST structure and relies on implementing the `IMayHaveEffect` interface and overriding its `effectDescriptor` method correctly. While this is simpler for the language implementor or extender, a drawback of this approach is an over approximation in one particular case: if you declare a function to take a function type that has an effect, then, even if a call passes a function without an effect, the call will still be marked as having an effect:

```
fun f*(g: ( =>* string)) = g.exec()* // declaration
f*(:noEffect)                        // call
```

Not Designed for Building Abstractions. KernelF is not optimized for building custom structural or behavioral abstractions. For example, it has no classes and no module system. The reason for this apparent deficiency lies in the layered approach to DSL design shown at the end of Sect. 1.2: the DSLs in which we see KernelF used ship their own domain-specific structural and behavioral abstractions. More generally, if sophisticated abstractions are needed (for example, for concurrency), these can be added as first-class concepts through language engineering in MPS.

There are also no algebraic data types. Option types and attempt types can be seen as a special case of algebraic data types, but we decided against implementing the general case for two reasons. The first reason is the general non-need for building abstractions. And second, by making attempt and option types first class, we can support them with special syntax and type checks (e.g., the `try` expression for attempt types) or by making an existing concept aware of them (the `if` statement wrt. option types).

Keyword-Rich. In contrast to the tradition of functional languages, KernelF is keyword-rich; it has relatively many first-class language constructs. There are several reasons for this decisions, the main reason being simplified analyzability: if a language contains first-class abstractions for semantically relevant concepts, analyses are easier to build. These, in turn, enable better IDE support (helping with [SIMPLICITY] and making the language easier to explore for the DSL users) and also make it easier to build generators for different platforms ([PORTABILITY]) Finally, in contrast to languages that do not rely on a language workbench, the use of first-class concepts does not mean that the language is sealed: new first-class concepts can be added through language extension easily.

3.2 Extension and Embedding

Here is a quick overview of the typical approaches used for extension of KernelF. We illustrate all of them in our case studies in Sect. 5.

Abstract Concepts. A few concepts act as implicit extension points. They are defined as abstract concepts or interfaces in KernelF, to enable extending languages to extend these concepts. They include `Expression` itself, `IDotTarget` (the right side of a dot expression), `IFunctionLike` (for function-like callable entities with arguments), `IContracted` (for things with constraints) and `Type` (as the super concept of all types used in KernelF). `IToplevelExprContent` is the interface implemented by all declarations (records, functions, typedefs).

Syntactic Freedom. A core ingredient to extension is MPS' flexibility regarding the concrete syntax itself: tables, trees, math or diagrams are an important enabler for making KernelF rich in terms of the user experience.

KernelF is Modular. The language itself is modular; it consists of several MPS languages that can be (re-)used separately, as long as the dependencies shown in Fig. 1 are respected. Importantly, it is possible to use only the basic expressions (`base`), or expressions with functional abstractions (`lambda`). Nothing depends on the `simpleTypes`, so these can be replaced by a different set of primitive types (discussed below). We briefly discuss the dependencies (other than those to `base`) between the languages and explain why they exist and/or why they do not hurt:

- **A:** required because of higher-order functions (`where, map`) on collections
- **B:** `path` navigation usually also has 1:n paths, which requires collections
- **C:** `repl` is a utility typically used when developing larger systems, which usually also use `toplevel` expressions; so the dependency does not hurt.
- **D:** `tests` are themselves top level elements; also, a dependency on `toplevel` does not hurt for a *test* model.
- **E:** functions in `toplevel` require generic function-like support from `lambda`
- **F:** the transactions in `mutable` require the blocks from `lambda`.

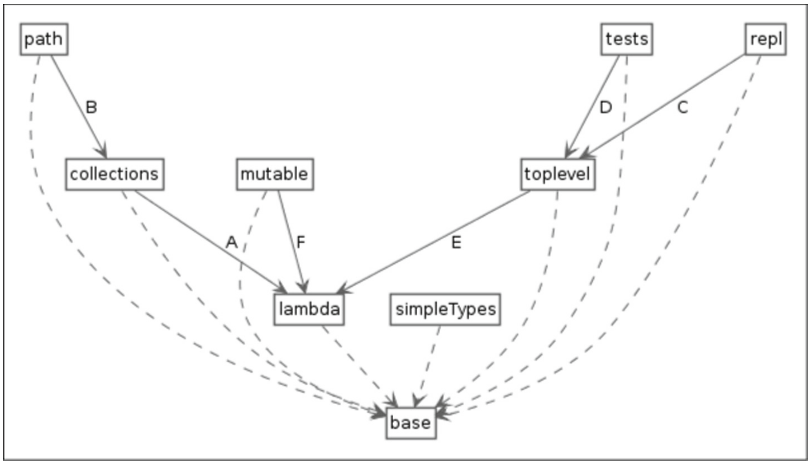


Fig. 1. Dependencies between the language modules in KernelF.

Removing Concepts. In many cases, embedding a language into a host language requires the removal of some of the concepts from the language. One way of achieving this is to use only those language modules that are needed; see previous paragraph. If a finer granularity is needed the host language can use constraints to prevent the use of particular concepts in specific contexts. A concept whose use is constraint this way *cannot be entered* by the user – it behaves exactly as if it were actually removed from the language.

Exchangeable Primitive Types. Many DSLs come with their own primitive types, so it is crucial that it is possible to *not* use `kernelF.primitiveTypes` when KernelF is embedded into a particular DSL. Preventing the user from entering a particular type into the program can be achieved with the approach described in the previous paragraph. However, the type system rules in the `kernelF.base` language rely on primitive types (some built-in expressions must be typed to Boolean or integer). This means that the types constructed in those rules types must also be exchangeable. To make this possible, KernelF internally uses a factory to construct primitive types. Using an extension point, the host language can contribute a different primitive type factory, thereby completely replacing the primitive types in KernelF.

Structure vs. Types. The types and the underlying typing rules can be reused independent from the language concepts. For example, if a language extension defines a its own data structures (e.g., a relational data model), the collection types from KernelF can be used to represent the type of a `1:n` relation.

Scoping. Scopes are used to resolve references. Every DSL (potentially) has its own way of looking up constants, functions, records, typedefs or its own domain-specific declarations. To make the lookup strategy configurable, KernelF provides an interface `IVisibleElementProvider`. Host language root concepts can implement this interface and hence control the visibility of declarations.

Overriding Syntax. Imagine embedding KernelF into a language that uses German keywords: the keywords of KernelF must now be adapted. MPS' support for multiple editors for the same concepts makes this possible.

4 Evolution

Number Types. Initially, KernelF had been designed with the usual types for numbers: `int` and `float`. However, even in our very first customer projects it turned out that those numeric types are really too much focussed on the need of programmers (or even processors), and that almost no business domain finds those types useful. Thus we quickly implemented the number types as described earlier. Since this happened during the first real-world use, this evolution did not involve any migration of existing, real-world models of customers, making the evolution process very simple.

Transparent Options and Attempts. Initially, option types and attempt types were more restrictive than what has been described in this paper. For example, if a value of `option<T>` is expected, users had to return `some(t)` instead of just `t`. Similarly for attempt types: users had to return a `success(t)`. Options and attempts also were not transparent for operators. For example, the following code was illegal, users first had to unpack the options to get at the actual values, which lead to hard to read nested `if` expressions.

```
val something : opt<number> = 10
val noText    : opt<string> = none
something + 10 ==> 20   <option[number[-inf|inf]{0}]>
noText.length  ==> none <option[number[0|inf]{0}]>
```

The reasons for the initial decision to do it in the more strict way were twofold. One, we thought that the more explicit syntax would make it clearer for users what was going on (less magic). Instead it turned out it was perceived as unintuitive and annoying. The second reason was that the original explicit version was easier to implement in terms of the type system and the interpreter, so we decided to go with the simpler option.

The migration to the current version happened after significant end-user code had been written, and so we implemented an automatic migration where possible: all `some(t)` and `success(t)`were replaced by just `t` by migration script that was automatically executed once users opened the an existing model once the new language version was installed. The unnecessary unpackings were flagged with a warning that explained the now possible simpler version. We expected users to make the change manually because we were not able to reliably detect and transform all cases, and because automated non-trivial changes to users' code is often not desired by users.

Enums with Data. Originally, enums were available only in the traditional form, i.e., without associated values. However, it turned out that one major use case for enums was to use them almost like a database table, where the structured value of one enum literal would refer to another enum literal (through using tuples or records as their value type):

```
enum T<TData> {
  t1 -> #TData(100, true,  u1)
  t2 -> #TData(200, false, u2)
  t3 -> #TData(300, true,  u2)
}

enum U<number> {
  u1 -> 42
  u2 -> 33
}
```

Records. According to our own design goal to keep KernelF small and simple, and in particular, the assumption that the host language would supply all (non-primitive) data structures, we originally did not have records. However, it turned out that this was a bad idea: records are useful as temporary data structures, even if the hosting DSL defines the notion of a component, class or insurance contract. Records are also useful for testing many other language constructs. However we did not add advanced features to records, such as inheritance; we reserve such features for host language domain-specific data types.

The internal implementation for records is based on interfaces. This way, it is very easy for extension developers to create their own, record-like structures that, for example, use custom syntax or support features such as inheritance. This extension hook has been used in several KernelF-based DSLs by now.

Range Qualifiers. A very common situation is to work with ranges of numbers. With the original scope of KernelF, for example, one could use an `alt` expression to compute a value `r` based on slices of another value `t`:

```
val r = alt | t < 10           => A |
            | t < 10 && t < 20 => B | // or t.range[10..20]
            | t > 20           => C |
```

However, as our users told us, this is perceived as unintuitive. The situation gets worse once uses range checks as part of decision tables, where many more such conditions have to be used. Our solution to this approach was to create explicit range qualifiers, so one could write the following code:

```
val r = split t | < 10   => A |
                | 10..20 => B |
                | > 20   => C |
```

These are not really expressions, because, for example `< 10` does not directly specify on which value the check has to be performed; that argument is implicit from the context. This is why these range qualifiers can only be used under expressions that have been built specifically for use with range qualifiers. The `split` expression is an example. We decided to make this part of the core KernelF language instead of an extension because these constructs are used regularly.

Enhanced Effects Tracking. Originally, there was only one `effect` flag: an expression either has an effect or it does not. However, when extending KernelF with mutable data, it became clear that we must distinguish between `read` and `modify` effects because, for example, a function's precondition or a condition in an `if` is allowed to contain expression that have read effects, but it is an error for them to have write effects. Interpreting "has effect" as "has modify effect" also does not work, because, even for expressions with read effects, caching is invalid.

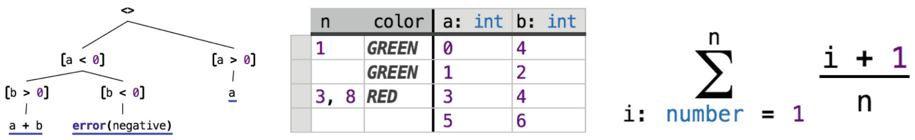
So far we have decided not to distinguish further between different kinds of effects (IO, for example), because this distinction is irrelevant for our main use of effect tracking, namely caching in the interpreter.

Mutable State. The initial plan for KernelF was to build a purely functional language and leave all state handling to extensions. While this is still fundamentally the case, it turned out that a general framework for dealing with state (beyond the declaration of effect discussed in the previous paragraph) is useful. In particular, `box`es enable the use of all functional/immutable data structures in a mutable way, and transactions handle the coordinated modification of multiple box-style values. The functionality is implemented as a framework (with interfaces such as `IBoxValue` or `ITransactionalValue`), and even if DSLs define their own abstractions and syntax for dealing with state, the use of those interfaces joins it together in a common semantic framework. This is why the `kernelf.mutable` language extension is now part of KernelF.

5 Case Studies

In this section I will present languages we built that extend and/or embed KernelF. Basically, they are all used in real-world customer projects, even though I took some liberty in assigning features to languages to make the discussion here more compact. We will discuss three of them in detail in the next subsections.

Utilities. A reusable language extension that supports decision tables of various shapes (actually rendered as tables), decision trees (actually rendered as trees), math notation (sum symbols, fraction bars, roots). Examples are in Fig. 2. All of these are `Expressions` and can (and are) used in many different languages. The language also supports range specifiers (`> 3` `4..8`) as well as type tags (useful to, for example, track tainted data or required confidentiality levels, as in `fun publish(d: Data<!secret>, receiver: Address)`).



n	color	a: int	b: int
1	GREEN	0	4
	GREEN	1	2
3, 8	RED	3	4
		5	6

$$\sum_{i:\ \text{number}\ =\ 1}^{n} \frac{i + 1}{n}$$

Fig. 2. `utils` extension: decision trees, multi-valued decision tables and math symbols.

Solver Language. Many language concepts benefit from various checks with a solver. For example, the decision trees and tables mentioned above can be checked for completeness and overlap-freedom. To simplify the integration of the solver with (domain-specific) language constructs, we have built an intermediate language that abstracts over the solver API. It provides was of defining constrained variables, as well as typical tasks for the solver, such as checking completeness, consistency, equality, progressive refinement or subsetting of expressions. The intermediate language itself makes use of KernelF to represent the expressions, but uses different primitive types.

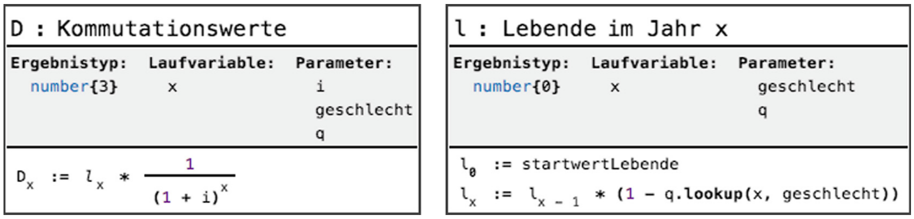
Healthcare. Voluntis' mobile apps help patients with therapies and treatments. The apps let users log data and they recommend actions such as taking a medication of a particular dose, behaving in particular ways or calling their medical team. The algorithms in these apps are "programmed" by doctors and healthcare professionals (HCP) using a KernelF-based DSL. The language reuses decision tables and trees and supports component-based behavioral modules, in particular, state machines. A second language supports expressing test and simulation scenarios. We discuss this language in detail in Sect. 5.3.

Salary/Tax Calculation. The purpose of this language is the specification of algorithms for salary and tax calculations based on German law. We have build extensions for ER-style data modeling as well as for calculation rules that recompute the data in a reactive way. The calculation rules and other declarations can be polymorphic regarding their validity periods (the `tax` must be calculated with rule `A` between until 2017, and then using rule `B` from 2018 onwards). Finally, the language support temporal arithmetics, with operators overloaded to work with data whose value changes over time. Details about this language are presented in Sect. 5.1.

Smart Contracts. We have developed a set of language extensions for efficiently and reliably defining smart contracts that emphasize multi-party collaborative processes. The language extensions comprise state machines (which are not specific to smart contracts), declarative abstractions for multi-party decisions, agreements and auctions, as well as ways of declaratively preventing several game-theoretical attack scenarios. This language also relies on boxes and transactions to manage a contract's state. Section 5.2 provides details.

Public Benefits. This system uses form-style syntax with embedded KernelF expressions to let legal experts formalize German public benefits law (unemployment payments, social welfare, old-age care support). In addition to the forms, the system has domain-specific expressions for representing idioms in public benefits payments. Finally, systematically representing the variability in law between Germany's 16 states is another challenge for which this language provides custom-built abstractions and syntax.

Insurances. Insurance mathematicians use many conventions when writing down there heavily numerical, recursive functions. For example, they distinguish between iterator variables and parameters, where parameters remain constant in (recursive) calls to functions that declare the same parameters (see Fig. 3). Sameness is established by relating them to a common data dictionary definition, which is why the parameters do not declare types when they are used in functions; those types are in the data dictionary. The language also relies heavily on various forms of lookup tables.



```
D : Kommutationswerte
Ergebnistyp:  Laufvariable:  Parameter:
  number{3}      x              i
                                geschlecht
                                q
```
$$D_x := l_x * \frac{1}{(1 + i)^x}$$

```
l : Lebende im Jahr x
Ergebnistyp:  Laufvariable:  Parameter:
  number{0}      x              geschlecht
                                q
```
$$l_0 := \text{startwertLebende}$$
$$l_x := l_{x-1} * (1 - \text{q.lookup(x, geschlecht)})$$

Fig. 3. Definition of numerical, iterative insurance math formulas. Notice the calls to `l` and `D` that pass the parameters implicitly. The type of `q` as defined in the data dictionary (not shown) is a lookup table, which is why the `lookup` method is available.

Cloud-Based App Development. Our customer uses a proprietary object-oriented programming language to develop and customize cloud-based applications. The language provides first-class support for their particular style of UIs and persistence layer. KernelF is used as the functional core, the object-oriented abstractions and a module system is built around it. Execution is based on their own, existing cloud-based interpreter infrastructure, so KernelF (and their embedding language) is transformed to their interpreter's byte code format.

Systems Engineering. Several customers use MPS-based DSLs for systems engineering, focusing on different aspects (such as structural modeling, performance prediction, and security analysis). All reuse a common, hierarchical component modeling language and a feature modeling language, both rendered in their natural graphical notations. KernelF expressions are embedded in various places, to define define type constraints on interfaces, to compute aggregate attribute values, to propagate configuration values and to navigate over component structures.

Meta Languages. As part of the Convecton[3] project, a new browser-based language workbench, we have developed a set of new meta languages which all rely on KernelF regarding their functional core. The interesting challenges here is the delineation between expressing behaviors functionally and domain-specific declarative abstractions.[4] The former are straight forward to build (and debug), but the latter have advantages in terms of forward execution (for example, to automatically derive quick fixes for errors). The code below illustrates a scope definition that determines the valid targets for a reference. Note how it separates the language feature (`from`) and `path` from the `filter` that selects targets; the former two can be reused for the `create` parts.

```
scope FunCall::function -> pick from Module::declarations
                                path (node, prnt) = node.container<Module>.imported()
                              filter (node, prnt, candidate) = candidate.isPublic()
                              create (node, prnt, futureParent, prefix) = mkLabel(prefix)
                                  at (node, prnt) = before(node.ancestor<Declaration>)
```

5.1 Salary/Tax Calculation

For this project, we have created several languages, as shown in Fig. 4. `date` is a language or representing dates and some of their arithmetics. `currency` contains types, literals and arithmetics for working with EUR currency. `data` provides entities and their relationships. The core of the system is in the `temporal` and `calculation` extensions. `temporal` contains temporal data types, literals and arithmetics, and `calculation` contains to-be-computed data structures as well as the rules to calculate them. In particular, the language supports the evolution of calculation rules over time, a core feature for representing the changing tax law. We discuss each of these languages in the remainder of this section.

Date Types. The system has to deal a lot with dates: people get married at particular dates, their salary changes at particular points in time and a salary calculation is valid for a particular month. So we need data types for dates, plus arithmetic operations for adding time periods to dates or finding the number of days between two dates. In the `date` extension we introduced a `date` type,

[3] http://convecton.io.
[4] https://languageengineering.io/thoughts-on-declarativeness-fc4cfd4f1832.

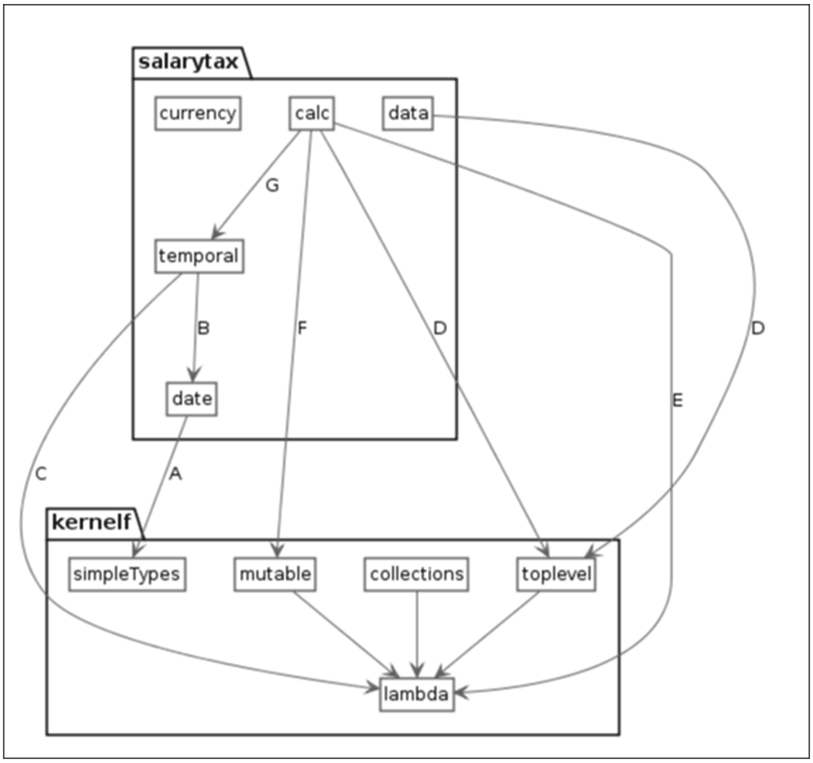


Fig. 4. Overview over the languages created in this system and their dependencies. The reasons for the dependencies are as follows: A: mixed arithmetics between dates and numbers; B: temporal slices use `date` values; C: temporal values support higher-order operations that contain lambdas; D: both contain top level declarations; E: treats the `BlockExpression` specially; F: requires `ILValue` for data field assignments; G: special typing rules for dealing with temporal values in calculation rules. Note that all of them depend on `kernelf.base`; the dependency has been elided to declutter the diagram.

as well as date literals, written as `/yyyy mm dd/`. They can be used like any other primitive type in KernelF, and the literals are expressions whose type is `DateType`, the concept behind `date`. The following is then valid:

```
fun printDate(d: date) {...}
val today = /2018 01 23/  // date type inferred
{ printDate(today) }
```

The reason for the unusual notation for date literals is to retain [WRITEABILITY] despite a particular drawback of the projectional editor: eager binding. If we were to use `yyyy/mm/dd` then, when you enter the slash behind the year, MPS interprets this as a division binary operator. Since there is no context by which to distinguish these two cases, the user would have to disambiguate manually, which is tedious. We could use the German notation: `dd.mm.yyyy`. Even though `dd.mm` would be initially interpreted as a number with a decimal point, entering the second dot could be used to trigger a further transformation to a date literal. However, using the `/yyyy mm dd/` notation is just the simpler solution, despite its slightly worse [READABILITY] and domain alignment.

We have overloaded a few operators to work with date types, in particular `+` and `-`. The former can be used to add days (the base unit of time in this system), and the latter can be used to compute the number of days between two dates, i.e., to subtract two dates. The following is valid:

```
val nextWeek: date         = today + 7
val lastYear: date         = today - 365    // ignoring leap years for now :-)
val howLongIsAYear: number = lastYear - today
```

To make this valid KernelF, no structural changes are required, since the operators already exist. However, the type system and the interpreter have to be adapted. Both of these, however, can be done modularly, in the `date` language. For the type system, we add a new overloaded operations rules, an MPS concept that supports polymorphic typing, typically used with operators:

```
overloaded operations for PlusExpression
  left argument  :==: <date>
  right argument :<=: <int>
  result type    { <date> }

overloaded operations for MinusExpression     overloaded operations for MinusExpression
  left argument  :==: <date>                    left argument  :==: <date>
  right argument :<=: <int>                     right argument :==: <date>
  result type    { <date> }                     result type    { <number> }
```

MPS executes overloaded operations by searching for all of those contributed by the set of languages used in a particular model, and then executing the first one that matches; since core KernelF has no rules that involve date types, the ones defined by the extension language apply.

The interpreter extension works in a similar way: we define a new interpreter that lives in the `date` extension that contains the two evaluators for `PlusExpression` and `MinusExpression`. Both first perform a check of the types of the arguments, and if they don't fit, return `tryAnotherInterpreter`, which triggers the interpreter framework to continue its search for a matching evaluator. Otherwise we use the JDK's date API for the respective arithmetics.

Currency Types. Another primitive type we have introduced for this system is `currency`. It is fundamentally a number with two decimals, the literals are written as `NN.DD EUR`. Their implementation is essentially identical to the date types discussed above, so we do not discuss it any further.

Temporal Types. A more interesting extension concerns temporal types. The notation `TT[U]` represents a temporal version of a base type `U`. Temporality means that a variable `ttu: TT[U]` does not represent a single value; instead, `ttu` is a sequence of `(date, U)`-pairs, expressing when the particular value of `ttu` changed to a particular `u: U`. The following example states that on Jan 1, 2017 the `salary` became 5.000 EUR, and on May 1 it changed to 6.000 EUR.

```
val salary : TT[currency] = TT | /2017 01 01/  =>  5.000 EUR |
                               | /2017 05 01/  =>  6.000 EUR |
```

The reason for adding temporal types is that this customer's system is bitemporal [16], which means that the system manages two dimensions of time for each data item. The first one represents a data item's evolution over time, also known as its *validity time*. The above `salary` is an example, and it is readily obvious why this is useful: almost all quantities in (database-style) systems change as time passes. Representing this as a first class concept in a language makes computation with these values simpler, as we shall see. The second dimension of time is the *transaction time*, i.e., the time at which something became known to the system (and was stored). In a bitemporal system, the database stores both.

```
val salary#/2017 10 07/ = TT | /2017 01 01/  =>  5.000 EUR |
                             | /2017 05 01/  =>  6.000 EUR |
val salary#/2017 11 05/ = TT | /2017 01 01/  =>  5.000 EUR |
                             | /2017 05 01/  =>  5.500 EUR |
```

The example here essentially says that, on Oct 7, 2017, we knew that the salary was as in the previous example; but on Nov 05 we changed the second value to 5.500 EUR; we probably corrected a mistake. The database now contains both states of knowledge, the one from October, and the one from November. A typical use case in the context of our customer's system is to calculate the resulting tax for both perspectives, and then issue compensating transactions. In the example, the person would probably get some money back.

A fully bitemporal system is quite complex, not just in terms of the database and the implementation, but also from the perspective of the user, i.e., the person who uses the DSL to create the salary/tax calculation rules. This is why, in the interest of [SIMPLICITY], we only represent the first dimension (validity time) in the DSL programs, and handle the second one as part of a surrounding framework; we will not discuss it any further.

The temporal types support overloaded operators. Their most important characteristic is that they "reslice" the temporal periods according to what is shown intuitively in Fig. 5; except for the slicing, the semantics of the operators regarding the basic types remain unchanged.

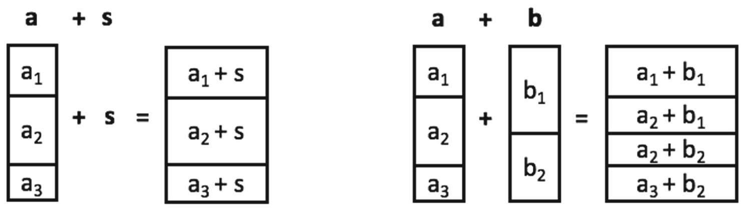


Fig. 5. Reslicing of temporal values; `a` and `b` are temporal values, `s` is a regular scalar. When a temporal value is "operatored" with a scalar, the slices remain the same, but their values change. In the case of two temporal values, the slices intersect, and the values are computed per intersection.

In the implementation, we once again had to overload the typing rules for the operators, this time for `TemporalTypes`. In this overload we fell back on the typing rules of the base type. For an operator `op`, `TT[U] op TT[V]` is allowed if `U op V` is allowed by the existing KernelF type system. The interpreter was built similarly to the one for data types, except that the implementation of the arithmetics is more complicated. Lots of test cases helped us get it right.

The overloaded operators let users write arithmetic code that works with temporal data as if it was regular, scalar data. Being able to do this was one major goal of this extension. But to effectively work with temporal data, more support is required, as illustrated below (values beginning with `d` are dates, and values beginning with `tt` are temporal):

– `always(v)` transforms a value `v: U` into a temporal value `ttv: TT[U]` with exactly one slice that is dated to a predefined "beginning of time" date.
– `ttv.add(d, v)` adds a new slice to `ttv` begins at `d` and has the value `v`.
– `ttv.valueAt(d)` returns the scalar value at time `d`.
– `ttv.between(d1, d2)` cuts the slices to within the range `d1 .. d2`. In addition, `ttv.after(d1)` and `ttv.before(d2)` are also supported.
– `ttv.reduce(S, r)` where `r: daterange` (a type that represents time periods) reduces a temporal value back to a scalar. The operation takes into account the slices within the time period `r` (for example, the month for which taxes are calculated) and a reduction strategy `S`. The strategy includes `LAST` (the value of the last slice in `r`), `SUM` (sums up all slice values), and `WEIGHTED_AVERAGE` where the sum is weighted with the relative lengths of each slice value. We will see examples of `reduce` below.

Basic Data. To model the basic data with which the system works (employee, address, employer, employment), the language supports another extension for data modeling. It supports entities with members which have either primitive, temporal, or other entities as type. In the latter case, cardinalities can be specified, as well as whether the relationship is containment or reference. Constraints, i.e., Boolean validation rules, are supported for entities as well. The language supports a textual and a (fully editable) graphical notation that can be switched on demand. Since this is "just another entity language" we do not discuss it any further.

Result Data. Result data are part of the resulting salary or tax calculations. Once computed, they are persisted in the database. The result data structures are similar to basic data entities in that the have a list of members. However, they are different in two important ways: first, they are always keyed by one or more basic data entities. For example, a `SalaryCalculation` result data item is always associated with an `Employment` entity, or the `TaxBill` result is associated with a `Person` entity. Second, result data items are time-indexed (which is different from being temporal). A time index identifies a discrete point in time and is typically `year` or `month`: the `SalaryCalculation` is indexed `monthly`, and `TaxBill` is indexed `yearly`. The association with the basic data entity and the time index uniquely identify each result data record. In the end, it is the purpose of the system to compute all result data item for all valid entity/time combinations.

Calculation Rules. A calculation rule's purpose is to compute a result data item for a given entity/time pair; so each rule is thus associated with one result data item. The rule also declares which other result data items it uses in its calculation. Consider the following example:

```
result data [monthly] Salary {                result data [monthly] Tax {
  employment -> Employment // basic data        person -> Person // basic data
  amount     : currency                         amount : currency
}                                             }

calculation for Tax
  depends Salary foreach person.employments     // depends on Salaries of all employments
          as salaries                           // of the Tax bill's person
                                                // in the respective time
  calculate [monthly] {
    val factor = // do some weird tax math
    val total   := salaries.amount.sum        // sum up all salaries in current month
    amount      := total * factor             // populate fields of the result data item
    employment  := ctx.employment             // ctx is available in all calculations
  }
```

Here, the calculation of the `Tax` relies on the calculation of the `Salary`. More specifically, it depends on all `Salary` calculations for the current `Tax`'s person's employments. Because these dependencies are explicit, they can be exploited during the execution. They can be used eagerly, like a function call: when the user requests the `Tax` for a particular person and month, the corresponding calculation rule is triggered, which in turn, when it calls `s.amount`, triggers the calculation of the `Salaries`. While this style of execution is good enough for in-IDE testing with the interpreter, a scalable engine for the data center will work in a reactive style. If a data item is changed, the dependencies are used in the reverse direction, and all dependent, upstream data is recalculated and persisted. This way, data is accessible to the user without the calculation delay incurred by the functional style. This is an example of [PORTABILITY] in the sense that different execution engines with different requirements in terms of performance and scalability can use the same specification. Importantly, the dependencies can also take into account the time index. Consider the next example:

```
calculation for SalaryReport // data structure indexed to an Employment
  depends Salary             as s
          Salary[month.prev] as s_last
  calculate [monthly] {
    currentSalary    := s.amount
    lastMonthsSalary := s_last.amount
    delta            := s.amount - s_last.amount
  }
```

The monthly salary report contains data from the previous month, as a means of providing context for the employee. In the example above, the `SalaryReport` data structure that stores this difference, has a dependency on the current months's `Salary` and on the one from the previous month, expressed with a little sub-language for expressing dependencies that take time into account. Since it is declarative (not full expressions!), it can also be evaluated in reverse order; it works with the reactive execution engine.

The salary and the report calculation rules are also marked as `monthly`. This is automatically derived from the data structure, which is time indexed `monthly` as well. This way it is clear that the execution of the `Salary` calculation rule always happens for a given time period, or `increment` (a month in this example). This leads to various syntactic simplifications. Consider the following:

```
calculation for Salary
  depends ...
  calculate [monthly] {
    val e                = ctx.employment
    val totalHoursWorked = e.workedHours.reduce(SUM)
    val averageWage      = e.wage.reduce(WEIGHTED_AVERAGE)
    val religion         = e.person.religion.reduce(LAST, increment.year)
  }
```

As we have seen above, the `reduce` operator requires the specification of a `daterange`, the time period for which the reduction applies. Because we are in a time-indexed context (`monthly`), this time period is implicit (the particular month) and we do not have to specify it. However, it can be specified if we need a different time period, as shown in the religion example, where we want to get the last slice's value in the current increment's year. Not having to specify the date range explicitly helps with syntactic [SIMPLICITY], but also [ROBUSTNESS] because of the reduced potential for errors.

Note that in the code above, we use five KernelF extension languages together: the data language (the `employment` reference), the currencies (in the wages), the date extension (as part of the temporal types), the temporal types themselves as well as the main extension for result data and calculation rules. Except for an explicit dependency from temporal types to dates, there is no language-level coordination code (composite grammars, disambiguation logic); the extensions are independent, but still used together in the same program. Please refer to Fig. 4 to recap the dependencies between the various languages.

Variants and Validity. Calculation rules depend on result data items, not on particular calculation rules for items. This is because there can be many calculation rules for a single result data item. There are two primary reasons for this. First, different calculations might apply for various context, such as different states, for married or unmarried people or for weekly vs. monthly pay. Instead of making all of these distinctions with conditionals in one rule, we can define a set of rules where each rule declares its applicability up front. Conditionals vs. multiple rules allow different tradeoffs regarding modularity, understandability and duplication, and thus help with [SIMPLICITY] and [ROBUSTNESS]. The second reason is that the algorithms embodied by the rules change over time, usually because of changes in the law that forms the basis for the calculation. Thus, a calculation specifies applicability and validity:

```
calculation for Tax                calculation for Tax
  depends Salary as s                depends Salary as s
    valid from /2017 01 01/                  SomeOtherThing as t
  calculate [monthly] {                valid from /2017 07 01/
    ...                                   if ctx.employment.person.homeAddress.state == BW
  }                                  calculate [monthly] { .. }
```

In the example above, we define a generally applicable `Tax` calculation that is valid from Jan 1, 2017. From Jul 1, a special rule has to be used if the employee

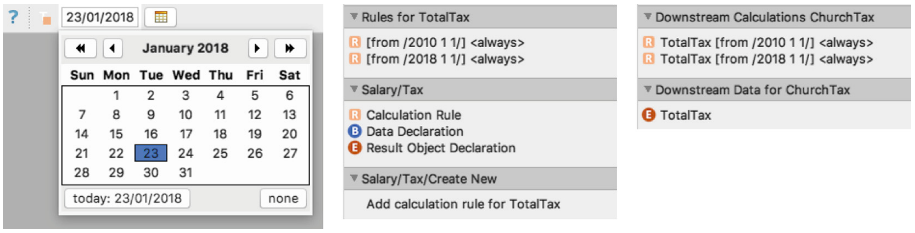

23/01/2018
January 2018
Sun Mon Tue Wed Thu Fri Sat
today: 23/01/2018
none
Rules for TotalTax
[from /2010 1 1/] <always>
[from /2018 1 1/] <always>
Salary/Tax
Calculation Rule
Data Declaration
Result Object Declaration
Salary/Tax/Create New
Add calculation rule for TotalTax
Downstream Calculations ChurchTax
TotalTax [from /2010 1 1/] <always>
TotalTax [from /2018 1 1/] <always>
Downstream Data for ChurchTax
TotalTax

Fig. 6. IDE features that support the language: date chooser to adapt the code to show only those parts that are valid at that date; context buttons for the currently selected result data item and for the currently selected rule.

lives in the `BW` state. If some other calculation rule declares a dependency on `Tax`, then, during execution, a dynamic dispatch will be performed that takes `valid` and `if` into account. The reason why this works is that all rules for a given result data item have the same signature (no arguments), so a transparent runtime dispatch is feasible – just as in object-oriented programming. However, the data structure can also change:

```
result data [monthly] Salary {          result data [monthly] Salary from /2017 10 01/ {
  employment -> Employment                employment -> Employment
  amount     : currency                   amount     : currency
}                                         taxFree    : boolean
                                        }
```

In this example, from Oct 1, we have to populate a Boolean flag that determines if that salary is tax free. In this case the IDE has to be aware of the new version, because the code that the user writes must now populate this field; instead of this version being a runtime dispatch only, it now has to be taken into account by the scoping rules and the IDE.

IDE Features. To keep track of the validity and applicability, we have implemented several IDE features, illustrated in Fig. 6. First, through a drop down box in the toolbar, users can optionally select a date for which they want to see the rules. If a date is selected, the editor evaluates the validity expressions and shows only those calculation rules that are valid at this point. In addition, if the user selects a data item in the editor, a palette shows all the rules that apply to this item. If the user selects a calculation rule, the palette shows the other rules for the same data item, as well as all (directly) downstream dependencies. There is a synergistic relationship between the language design and [IDESUPPORT]: if the applicability and validity were implemented as conditionals in the body of a rule, if would be *much* harder to provide this kind of tool support.

The IDE also helps with consistency. Since the validity is only specified using a `from` date, there is no need to check for consistency and completeness. However, we will implement a graphical timeline that shows how the periods of various rules and their dependencies align. However, there is one particular aspect that must be verified for the combination of runtime polymorphism (as used in calculation rules) and static polymorphism (as used for the result data structures).

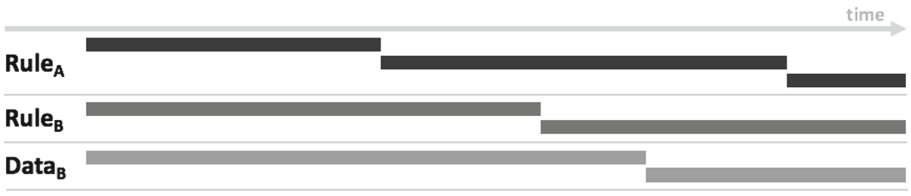


Fig. 7. Example of allowed (`Rule-A/Rule-B`) and non-allowed (`Rule-B/Data-B`) misalignment between validity periods.

Consider the scenario in Fig. 7. Assuming that `A` depends on `B`, it is *not* a problem that the validity periods for the variants for `Rule-A` and `Rule-B` do not fully overlap, because the runtime dispatch is transparent to the programmer. However, the validity period misalignment between `Rule-B` and its result data structure `Data-B` is an error because the same rule would have to work with different data structures, in the IDE. This is not possible.

5.2 Smart Contracts

Blockchains [22] and smart contracts promise trusted, distributed execution of arbitrary programs. Ethereum [32] is currently the most relevant platform as a consequence of its flexible VM, expressive languages, comparatively mature infrastructure and adoption rate. Several languages, all compiling to EVM bytecode, exist, the most widely used one is Solidity. Solidity[5] is essentially a general-purpose programming language that also has some features that are specific to Ethereum's VM and distributed execution model.

However, Solidity does not provide first-class support for the typical patterns found in the distributed, multi-party contrats for which blockchains are supposedly ideally suited. Such abstractions are critical if we consider that a lot of the interest in blockchains and smart contracts comes from non-technical people in domains such as finance [21], logistics [18] or (computational) law.[6] They are very likely also the people who are interested in the specific behaviors encoded in the contracts. So, while ensuring the correctness of the EVM and blockchain infrastructure is crucial [1,15], a concise, understandable and (functionally) verifiable specification of contracts is also crucial. The language introduced here has this goal, but is of course not the only [11] one.[7]

Figure 8 shows stack of languages that potentially achieves this goal. At the top we envision DSLs that are specific to the business domains for which contracts should be specified. We have some exposure to finance, logistics and law, and the requirements are quite different. On the level below we envision a language (dubbed EMPCL) that has the basic abstractions for executable multi-

[5] http://solidity.readthedocs.io/.
[6] https://www.artificiallawyer.com/2018/01/19/welcome-to-the-first-computational-law-blockchain-festival/.
[7] https://runtimeverification.com/blog/?p=496.

party collaborative processes. This language, in turn, extends and embeds KernelF. For execution, the contract behaviors are generated to suitable blockchain technologies, and for verification, an integration with model checkers and solvers is useful. In this chapter we focus on a prototypical implementation of EMPCL.

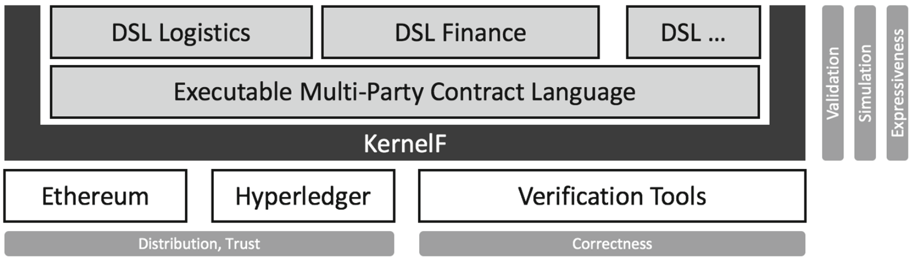


Fig. 8. A language architecture consisting of industry-specific DSLs, a set of common abstractions in EMPCL, the KernelF foundation, plus generation to blockchain-based execution infrastructures and verification tools for ensuring functional correctness.

Processes. A contract's evolution over time is inherently stateful. The work on smart contracts drove many of KernelF's extensions for stateful programs, such as boxes and transactions (see Boxes and Transactions in Sect. 2 as well as [25]). Before we illustrate those features, let me introduce the notion of a process. A process is a declarative description of a stateful, potentially long-running behavior. The process definition configures the behavior and determines how programs interact with it in terms of commands (that trigger changes in the process) and values (how the environment can observe the process state). Processes are a good baseline for representing the idiomatic behaviors expressed in Smart Contracts such as decisions, agreements or auctions. We performed a preliminary domain analysis for decisions and identified the following variations points: which parties are involved in the decision, and can that list of parties be changed dynamically during the execution of the decision process, what is the decision procedure (unanimous, majority, specific threshold or completely custom), is a minimum turnout required, is there a time limit for making the decision, and can votes by a particular party be revoked. Figure 9 shows the notation used for `MultiPartyDecisions`, and a few example configurations (ignore the code completion menus for now). Once defined this way, processes can be instantiated and used; the code below uses the leftmost process in Fig. 9.

```
val s = run(Unanimous)                          // continued
s.vote(bernd)                                   s.vote(markus)
s.vote(bernd)                                   assert(s.decisionTaken)
```

The process above has one command `vote(party)` and one Boolean value, `decisionTaken`. Which commands and values are available, depends on the configuration of the process. For example, if we were to configure `dynamic` parties, an additional command `addParty(party)` would be available (an example of

[IDESUPPORT]). This is interesting in two respects. From a language design perspective, the fact that available commands and values depend on the process configuration prevents the user from making certain mistakes; a degree of correctness-by-construction is guaranteed, helping with [ROBUSTNESS]. As a point of comparison, this feature could not be provided by an OO framework, because it requires an IDE's awareness of the program's semantics, specifically for the process abstraction.

Second, it is interesting from a language implementation perspective. Normally, a method call is an actual reference (in terms of the MPS AST) to the method declaration. Here, no method `vote` or `decisionTaken` is available to act as reference targets. This is why we have implemented a "reflective" mechanism for commands and values. The process declares and registers them with a descriptor, depending on the process's configuration:

```
final IDCommand VOTE = new IDCommand("vote", new IDArg("who", <PartyType()>));
final IDValue DEC_TAKEN_BOOL = new IDValue("decisionTaken", PTF.createBooleanType());
final IDCommand ADD_PARTY = new IDCommand("addParty", new IDArg("who", <PartyType()>));

public void populateDescriptor(ProcessDescriptor d) {
  d.add(VOTE);
  d.add(DEC_TAKEN_BOOL);
  if (this.dynamic) {  // this queries the dynamic flag in the process definition
    d.add(ADD_PARTY);
  }
}
```

The invocation syntax (`process.value` and `process.command(args)`) is also generic: the node on the right side of the dot is not a reference, as mentioned above, instead it only stores the string that represents the name of the value or command. Code completion proposes only those strings that correspond to the currently active values or commands on the target process, and the type checker also relies on the descriptors to check for valid names and arguments. A language user cannot tell the difference; it behaves exactly like "native" references.

Meta Functions. This is also a good place to demonstrate how to "escape from declarativeness": what to do if you want to provide a declarative means for configuring something (supporting [SIMPLICITY] for the simple cases), but still allow the option of injecting arbitrary code. We will illustrate this with the process' decision procedure: in terms of structure, the process has a child `proc` that is a `DecisionProcedure`, which is an abstract concept. I has three subconcepts: `UnanimousDecProc`, `MajorityDecProc` and `CustomDecProc`. The first two are just keywords, whereas the last one looks as follows:

```
procedure: custom (voted, participated) = voted.size > (2/3) * participated.size
```

The custom decision procedure embeds a meta function. A meta function has a number of parameters as well as an expression that computes a value from the parameters. Meta functions are a generic utility, they can be configured and executed easily: in terms of structure, `CustomDecProc` only has to implement `IMetaFunctionContext`. In its behavior, it overrides the `createMetaFunction` method to create the meta function structurally; in particular, it specifies the name, return type and arguments:

```
public node<MetaFunction> createMetaFunction()
  createNew(PTF.createBooleanType(), "custom␣procedure")
    .addArg("voted", <ImmutableSetType(baseType: PartyType())>)
    .addArg("participated", <ImmutableSetType(baseType: PartyType())>);
}
```

Execution is just as straightforward. The `DecisionProcedure` declares a behavior method `isDecided` that returns true of false, depending on whether the decision has been made or not. The custom procedure implements it as follows:

```
public boolean isDecided(PSet parties, PSet whoVoted,
                         IContext ctx, ComputationTrace trace)
  (boolean)(new MFI(ctx, this.function).run(whoVoted, parties, trace));
}
```

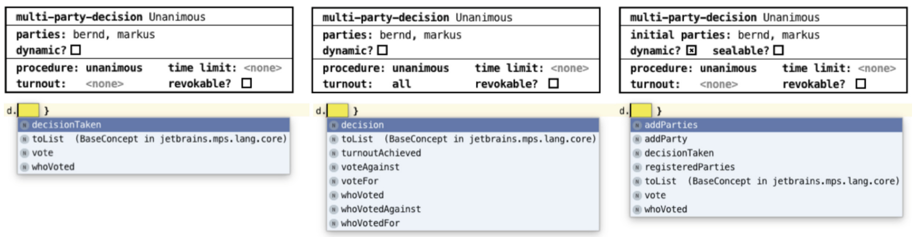
This code instantiates the meta function interpreter (`MFI`), passing the interpreter context and the to-be-executed function (the `function` child is inherited from `IMetaFunctionContext`). Calling `run`, we pass values for the two arguments defined for the function, `voted` and `participated`. The return value is the Boolean flag that indicates whether the decision is successfully taken or not.

State, Boxes, and Transactions. The primary benefit of boxes is that existing immutable data structures and their APIs can be reused in a mutable way in the sense that the box stores an evolving sequence of immutable values. All immutable data structures can immediately be used this way. In addition, boxes allow a straightforward implementation of transactions:

– The user marks the start of a transaction in the program code; a `Transaction` object is put into the interpreter context
– For any `update` of a box, the new value is stored in a `map<box, value>` that lives inside the transaction object; the box contents are not actually modified.
– Inside a transaction, a box read is redirected to a lookup in the map[8] (which we can find out by looking for a `Transaction` object in the context)
– When we commit the transaction, the actual box contents are updated based on the map inside the `Transaction`
– If the transaction is cancelled (for whatever reason), the map is discarded and the boxes stay unchanged

There are also language constructs that make sense only in a stateful context. The processes introduced above, as well as the state machines we will discuss below, are examples. For them, there is no point in defining an immutable API, and consequently there is also no benefit in using boxes to be able to reuse an immutable API in a mutable context. This is why the decision's `vote` or `addParty` commands directly change the state of the process; they are a mutable API. However, internally, objects that are mutable in this way still rely on immutable data. In other words, a change to the state of the process internally sets a new, updated state object.

[8] Note that this also works if multiple transactions run in a concurrent context; isolation is maintained because the boxes themselves are not updated.



```
multi-party-decision Unanimous
parties: bernd, markus
dynamic? ☐
procedure: unanimous      time limit: <none>
turnout:    <none>        revokable? ☐
d.| }
  decisionTaken
  toList  (BaseConcept in jetbrains.mps.lang.core)
  vote
  whoVoted
```

```
multi-party-decision Unanimous
parties: bernd, markus
dynamic? ☐
procedure: unanimous      time limit: <none>
turnout:    all           revokable? ☐
d.| }
  decision
  toList  (BaseConcept in jetbrains.mps.lang.core)
  turnoutAchieved
  voteAgainst
  voteFor
  whoVoted
  whoVotedAgainst
  whoVotedFor
```

```
multi-party-decision Unanimous
initial parties: bernd, markus
dynamic? ☒  sealable? ☐
procedure: unanimous      time limit: <none>
turnout:    <none>        revokable? ☐
d.| }
  addParties
  addParty
  decisionTaken
  registeredParties
  toList  (BaseConcept in jetbrains.mps.lang.core)
  vote
  whoVoted
```

Fig. 9. A couple of different configurations of the `multi-party-decision` process and the resulting available entries in the code completion menu.

```
public void handleCommand(IDCommand command) {
  if (command.is(VOTE)) {
    string party = (string) payload.first;
    state = state.voteFor(party); // old state is cloned with a new vote
  }
  ...
}
```

Effectively, this makes a process (and other similar construct) a kind of "implicit box". Explicit and implicit share the runtime API through which they interact with a transaction. This way, they can be used together:

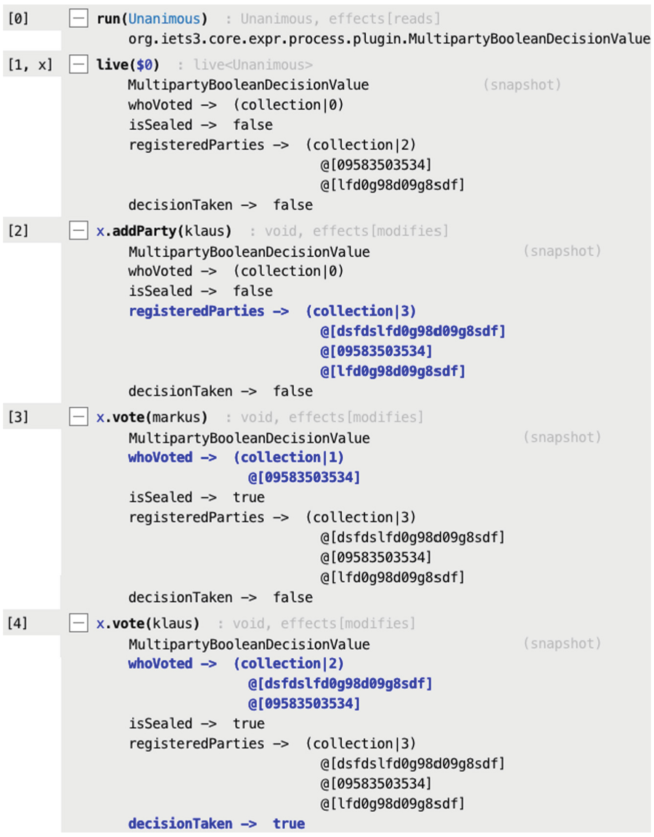
```
val voteCount = box(0)                        fun voteAndCount(Party whoVotes)
val process = run(Unanimous)                    newTx { process.vote(whoVotes)
                                                        voteCount.update(it + 1) }
```

When calling the transactional function `voteAndCount`, and if the `vote(whoVotes)` fails (for example, because the party `whoVotes` is not a valid voter), then both the `process` and the `voteCount` remain unchanged.

Live Values and the REPL. The values published by the processes provide a peek into its internal state, based on a generic, reflectively-defined API. In addition, using a `LiveValue` wrapper, the processes provide these values in a structured way, suitable for display in the REPL. Because of this homogeneous structure, the REPL highlights the diffs between the current state and the previous one. Figure 10 shows a REPL session.

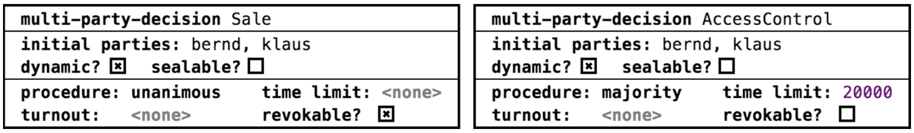
The generic interaction mechanisms of processes (and their generalised version, `IInteractors`) are a good starting point for building simulators or other end-user oriented UIs (roughly similar to [4]). For example, commands can rendered as buttons, and the values can be rendered as text labels or other widget. Because the internal state is a sequence of mutable values that can be retained in such a simulator, it is easy to build "time travel" functionality [3] or even branching, where users can interactively explore back and forth the behaviors of contracts.

More Complex Contracts. We decided on declarative abstractions for the core decision, agreement, auction and sales processes because those are ubiquitous in smart contracts. In some sense they can be seen as the building blocks of contracts. In addition, it is feasible to capture the vast majority of real-world



```
[0]     run(Unanimous)  : Unanimous, effects[reads]
          org.iets3.core.expr.process.plugin.MultipartyBooleanDecisionValue
[1, x]  live($0)  : live<Unanimous>
          MultipartyBooleanDecisionValue                  (snapshot)
          whoVoted -> (collection|0)
          isSealed -> false
          registeredParties -> (collection|2)
                                 @[09583503534]
                                 @[lfd0g98d09g8sdf]
          decisionTaken -> false
[2]     x.addParty(klaus)  : void, effects[modifies]
          MultipartyBooleanDecisionValue                  (snapshot)
          whoVoted -> (collection|0)
          isSealed -> false
          registeredParties -> (collection|3)
                                 @[dsfdslfd0g98d09g8sdf]
                                 @[09583503534]
                                 @[lfd0g98d09g8sdf]
          decisionTaken -> false
[3]     x.vote(markus)  : void, effects[modifies]
          MultipartyBooleanDecisionValue                  (snapshot)
          whoVoted -> (collection|1)
                        @[09583503534]
          isSealed -> true
          registeredParties -> (collection|3)
                                 @[dsfdslfd0g98d09g8sdf]
                                 @[09583503534]
                                 @[lfd0g98d09g8sdf]
          decisionTaken -> false
[4]     x.vote(klaus)  : void, effects[modifies]
          MultipartyBooleanDecisionValue                  (snapshot)
          whoVoted -> (collection|2)
                        @[dsfdslfd0g98d09g8sdf]
                        @[09583503534]
          isSealed -> true
          registeredParties -> (collection|3)
                                 @[dsfdslfd0g98d09g8sdf]
                                 @[09583503534]
                                 @[lfd0g98d09g8sdf]
          decisionTaken -> true
```

Fig. 10. A REPL session where a processes is wrapped in a `LiveValue` to support structured rendering of the internal state and diffs that highlight its evolution.



```
multi-party-decision Sale
initial parties: bernd, klaus
dynamic? [x]   sealable? [ ]
procedure: unanimous     time limit: <none>
turnout:   <none>        revokable? [x]
```

```
multi-party-decision AccessControl
initial parties: bernd, klaus
dynamic? [x]   sealable? [ ]
procedure: majority      time limit: 20000
turnout:   <none>        revokable? [ ]
```

Fig. 11. Two decision processes used in the complex example contract.

variants into a set of configuration parameters. However, the overall contracts that make use of these building blocks show more variability, which is why it is more useful to use a less specific language for those: state machines are obvious candidates. Consider the following requirement for a non-trivial smart contract:[9]

An online community has to continuously maintain a (selling) decision; it can be revoked or granted as time passes. A group of individuals, called the

[9] Another extensively documented example can be found in [23].

deciders, vote for and against this decision. The vote has to be unanimous. In addition, additional people can be allowed into the group of deciders. The existing deciders vote for new candidates, by simple majority, but with a time limit. Once allowed into the group of deciders, the new member can participate in the sell/no-sell decision. Multiple member approval processes can go on at the same time. While a member request is pending, the sales decision cannot be changed.

The implementation of this contract relies on two declarative decision processes, `Sale` (to maintain the sales decision) and `AccessControl` (one is instantiated for each allow-in of a potential new decider). Both configurations are shown in Fig. 11. The remaining state machine-based implementation of this contract is as follows (we omit the `state machine` declaration itself). First, we define the events which we want to use to control the contract:

```
event openAccess  // go to the mode where we allow new guys to request to join
event requestAccess(newGuy: party) // a new guy wants to join the deciders
event terminateAccessRequest(who: party, newGuy: party) // kill a decision procedure
event voteForAccess(voter: party, newGuy: party) // vote for a new guy to become decider
event letsSell    // go to the state where we maintain the sell/no-sell decision
event voteForSelling(who: party)     // vote for the sale decision
event voteForStopSelling(who: party) // vote against the sale decision
```

Next, we instantiate one `Sale` process in the state machine, and define a map from `party` to `AccessControl` where we store all pending access requests. We also define a query (essentially a parameterless Boolean function) that reports whether the selling decision is currently true or false. The `observable` flag means that the query can be invoked from outside the state machine:

```
var sale                          = run(Sale)
var pendingAccess                 = box(map<party, AccessControl>())
observable query currentlySelling = sale.decisionTaken
```

The similarity between processes and state machines is not coincidental: in fact, the state machine also implements `IInteractor`, the events act as commands and the observable queries or variables correspond to values. Thus, state machines can be used in the same interactive way (for example in the REPL) as the processes in the previous paragraph.

Next we define a few helper functions used inside the state machine; the `/R` or `/RM` flags indicate the kind of effect they have (`read` only, or `read-modify`):

```
fun isDecider/R(who: party) = sale.registeredParties.contains(who)
fun isPending/R(who: party) = pendingAccess.val.keys.contains(who)
fun hasPending/R()          = pendingAccess.val.size != 0
```

The core logic is implemented in the next few states. The first one represents the phase where the contract is gathering new members. The following code handles the `requestAccess` event, where a new party can request access to the group:

```
on requestAccess(newGuy) [!isDecider/R(newGuy)] : {
  val acc = run(AccessControl)
  pendingAccess.update(it.put(newGuy->acc))
  acc.addParties(sale.registeredParties)
}
```

The transition only fires if the `newGuy` is not yet among the existing deciders (see guard condition); then we create a new `AccessControl` process and store it in the map that keeps track of the currently pending membership requests. Before that new `AccessControl` process can work, we have to populate it with the existing deciders, because it is them who make the decision about the membership of the `newGuy`. Note that this transition has no target state, so it remains in the current one; its only purpose is to perform the action associated with the transition.

The second transition terminates an existing access request if one of the deciders chooses to do so. The event has two arguments, the party who request termination and the party whose membership request should be terminated. The guard condition checks that these two parties actually play the respective roles. If everything is in order, we just delete the corresponding `AccessControl` process from the map of pending accesses.

```
on terminateAccessRequest(who, newGuy) [isDecider/R(who) && isPending/R(newGuy)]
   : pendingAccess.update(it.remove(newGuy))
```

Next we deal with a current member (`voter`) voting for a new guy. Again, we use the guard condition to establish the roles. We then get the `newGuy`'s `AccessControl` from the pending list and submit our vote. If after the voting the decision has been taken, we add the `newGuy` to the parties of our `Sale` process and remove their `AccessControl` from the list of pendings.

```
on voteForAccess(voter, newGuy) [isPending/R(newGuy) && isDecider/R(voter)] : {
  val acc = pendingAccess.val[newGuy]
  acc.vote(voter)
  if acc.decisionTaken then {
    sale.addParty(newGuy)
    pendingAccess.update(it.remove(newGuy))
  } else none
}
```

The last thing we do in the `requestAccess` state is to handle the request to move to the `selling` state, which is only possible if there are no pending requests (which is why current deciders can terminate pending requests by force):

```
on letsSell [!hasPending/R()] -> selling
```

The `selling` state is really simple. It handles voting for and against the sales decision maintained by the contract, as well as the `openAccess` event which gets us back into the state where we accept new members. Note how the actual logic of making the sales decision, independent of its own complexity, is handled completely by the `Sale` process.

```
state selling {
  on openAccess -> gatheringMembers
  on voteForSelling(who) [isDecider/R(who)] : sale.vote(who)
  on voteForStopSelling(who) [isDecider/R(who)] : sale.revoke(who)
}
```

Game Theory, Interceptors and Context Arguments. Game theory [12] looks at how rules in cooperative processes ("games") impact the outcome, and also how the parties taking part in the game can cheat, i.e., exploit the rules for their own benefit. Smart contracts are cooperative processes, which is why they are susceptible to "game-theoretical" exploits.

For example, a sybil attack [7] is one where a reputation-based system is subverted by one (real-world) party creating loads of fake (logical) identities who then behave in accordance with the real world party's goals. For example, consider a decision that is based on majority vote. An attacker could create lots of additional parties and thereby taking over the majority, leading to a decision in the interest of the attacker. While there are many potential ways how such attacks can be thwarted, one approach is to limit the rate at which new parties can request to join the process. Instead of requiring users to implement this manually, the state machine language supports a declarative way: the rate at which events come into a state machine can be limited (helping with [ROBUSTNESS] without compromising [WRITEABILITY]). The following code expresses that while the machine is in state `requesting`, only three commands per second are allowed. If more requests come in, they are rejected.

```
state requesting [rate(3/1000|commands-only)] {
  ...
}
```

The code between the brackets registers an interceptor (the term is inspired by CORBA [19]). Interceptors see every incoming event before transitions have an opportunity to react to them. They can then let it pass through, change parameters in the event, or discard it. Interceptors can maintain their own internal state. They can be seen as a guard condition that applies for a whole state (or substates), and not just a particular transition. The rate interceptor discards events if the rate exceeds the one specified.

Looking at the example, you can see that many events take the sender as an argument, usually in order to check that the event is authorised (the sender is among the current deciders). This is typical for smart contracts, and in fact, every message sent into an Ethereum contract carries an implicit sender address. Implicit arguments, called context arguments, are also available for interactors. Together with an interceptor, this can be used for authorization:

```
state playing [senderIs(players)] {
  on offerBid(money) : bids := bids.put(sender->money)
  ...
}
```

The `senderIs()` interceptor checks whether the context argument `sender` is supplied by the client (and rejects the event if not), and verifies that the sender is in the collection passed as an argument to `senderIs` (and rejects the event if not). In addition, because any transition in the state will only be executed if a sender is given, the interceptor makes the `sender` variable available inside the state. It can be used just like an explicitly given argument. In the example above we use it to create an entry in the `bids` map that is keyed by the `sender`.

The last interceptor worth mentioning in the context of smart contracts and game theoretical exploits is the `takeTurns` interceptor. Many "games" require a fair allocation of opportunities to participating parties. One way of achieving this is to run a game turn-by-turn, where each party can make one "move" in every "round". Consider the bidding process example:

```
state playing [senderIs(players)] {
  state bidding [takeTurns(players|ordered|after 1000 remove)] {
    on offerBid(money) : bids := bids.put(sender->money)
    if [timeInState > 2000] -> finished
  }
  ...
}
```

The `takeTurns` interceptor can be configured regarding the strictness of the turn-by-turn policy. `Unordered` means that in each round, every party has to make a move, but the order within each round is not relevant. `ordered` means that the order given by the list of parties passed to the interceptor is strictly enforced. A violation leads to a rejection of the command. The interceptor also provides access to the list of allowed next movers; this could potentially be used to notify parties that it is their turn.

There is a risk of a denial-of-service attack in the case of `ordered` turn taking: if the next party `p` does not make its move, the whole process is stuck. Nobody else can make a move because it is `p`'s turn. This is why a turn-by-turn game should always include a timeout, 1000 in the example above. If the next party does not make their move within 1000 time units, that party is permanently removed from the list of participants; alternatively, it can also be `skip`ped.

5.3 Healthcare

Like all the other case studies, the set of languages built for this system builds on top of KernelF and extends it with new expressions (see Fig. 12). High-level domain-specific behaviors are expressed as state machines, as explained below. However, this system is interesting because it removed about two thirds of KernelF (by constraining it out of program written in the context of this system). For example, attempt types, option types, some of the advanced operators as well as some of the collection operations are not accessible to the users of this system.

Reactive Algorithm. The main algorithm controls notifications and reminders submitted to the mobile operating system and reacts to a user's data submissions. It also makes high-level decision as to the execution of the algorithm and manages data collected from the user (in what one could call databases).

The top level abstraction is the component, a unit of behavior. Components can be instantiated and then started by other components, hierarchically. When a parent component starts a child component, it supplies values to the parameters defined by the child components (just like an operating system that starts a process). The child then runs concurrently with the parent; it communicates with the parent by sending output data events. The parent component can react

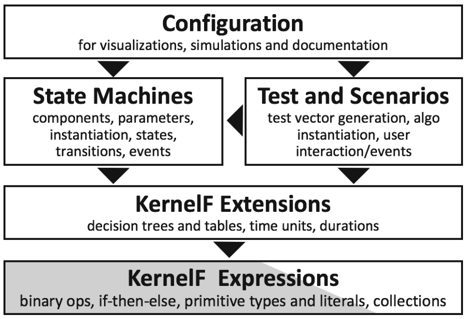

Configuration
for visualizations, simulations and documentation
State Machines
components, parameters, instantiation, states, transitions, events
Test and Scenarios
test vector generation, algo instantiation, user interaction/events
KernelF Extensions
decision trees and tables, time units, durations
KernelF Expressions
binary ops, if-then-else, primitive types and literals, collections

Fig. 12. The core of this system is a restricted version of KernelF. On top, we have developed a set of functional extensions that help medical professionals make non-trivial (multi-criteria) decisions. The core of the medical algorithm is expressed through state machines, and validation is performed through a testing and scenario description language. At the top is a language for configuring generated visualisations and reports.

to those events. By waiting for particular events (see below), the parent can synchronise with (wait for completion of) a child it started.

In addition, a component also provides other means of interaction with its environment, and in particular with the user, through the UI. A component can bring up a UI, for example, a questionnaire where the user can then select one of several options. A component can also register reminders: essentially, this is an entry in the phone's calendar. The framework that runs the applications on the phone keeps track of the created notifications, and retracts them if the user reacted, or if a timeout occurs.

The implementation of the behavior inside a component can potentially be done in many ways to be able to handle future styles of applications. For now, only a state-based implementation is supported: the content of a component is a hierarchical state machine. The abstractions are the usual ones: nested states, events, transitions, guards, actions.

Consider the following example: the application wants the user to measure their blood sugar at `08:00` the next morning. To this end, the application registers a reminder for `07:55`, `08:00` and `08:10`. Once a new blood sugar value is entered by the user at roughly 8am, the remaining reminders can be retracted. In contrast, if no value is entered by `08:10`, the process might have to react to that: for example, a message might have to be shown to the user reminding them of the importance of a timely blood sugar measurement, or, if things become more serious, their medical team might have to be notified by the app. To realize this behavior, a timeout event in the state machine is necessary.

The code to implement this behavior looks roughly like the following. We start by defining a helper function that computes the next time at which a blood sugar measurement should take place. The time literals, and the associated types use an addition data type `datetime` whose implementation is similar to the one defined in the salary/tax case study in Sect. 5.1[10]. Second, we define a `timeseries` for the blood sugar measurements. Time series are essentially records with an index of type `datetime`, and are also defined specifically for this system:

```
fun nextTime() {
  alt | now in [08:01 .. 11:45]  =>  12:00 |
      | now in [11:46 .. 17:45]  =>  18:00 |
      | otherwise                =>  08:00 |
}

timeseries BloodSugarSeries {
  value: number[50|400]
}
```

The meat of the blood sugar measurement functionality is in a component `AcquireBloodSugar`. It has two configuration parameters; the time at which the next measurement should take place, as well as the time series in which to store the measurement.[11]

```
component AcquireBloodSugar

parameters t : datetime
           db: BloodSugarSeries
```

Next, we define the interface of the component; it handles events of type `BloodSugarMeasurement` from the UI layer. In addition, it emits the `ok` and `missed` events, both without arguments.

```
inputs  BloodSugarMeasurement(bs: BloodSugar)

outputs ok
        missed
```

Next we declare a reminder. A reminder is essentially a group of OS-level reminders which, as we will see below, are managed as a group from the perspective of the algorithm. The reminders are defined relative to the time `t` passed to the component as a parameter.

```
val r = reminders at t - 15 : "Please enter blood sugar in 15 minutes"
                  at t - 5  : "Please enter blood sugar in 5 minutes"
                  at t      : "Please enter blood sugar now"
                  at t + 10 : "URGENT: Please enter your blood sugar"
```

Finally, we define the actual behavior of the component. Note that, because on mobile phones the app is passive when not in focus, and because it cannot actively push content to the user (except through reminders), the app is reactive. This is why a state machine is a very good fit. The `start` block is executed after the component is started (essentially a constructor). We create the reminders and unconditionally transition to the `waiting` state. In that state, as the name suggests, we passively wait for input events, i.e., the `BloodSugarMeasurement`. If one occurs, and the current time is within 20 min of the scheduled time `t`, then we store the measurement in the time series, and `terminate` with the `ok`

[10] We are currently consolidating both into a common datetime extension.

[11] The actual syntax relied a little bit more on boxes and other semi-graphical elements; we use text here so we do not have to resort to images.

event (`terminate(<evt>)` is a shorthand for sending an event (`send(<evt>)`) and then just terminating the execution of the component). If we do not receive the event within `t + 20`, we terminate with a `missed` event. In any case, once the `waiting` state is left, the OS-level reminders associated with `r` are all cancelled.

```
start: createReminders(r)
       -> waiting

state waiting:
  exit: cancelReminders(r)
  on BloodSugarMeasurement
    when now in [t - 20 .. t + 20]
      store now, bs in db
      terminate(ok)
    when now < t - 20
      message "too early, please submit around {t}"
  if now > t + 20
    terminate(missed)
```

Here is the (simplified) main state machine for the diabetes application that uses the `AcquireBloodSugar` component above. It creates and starts the `Acquire-`ıBloodSugar in its `running` state. It then keeps track of the missed measurements and, if too many are missed, notifies the medical team.

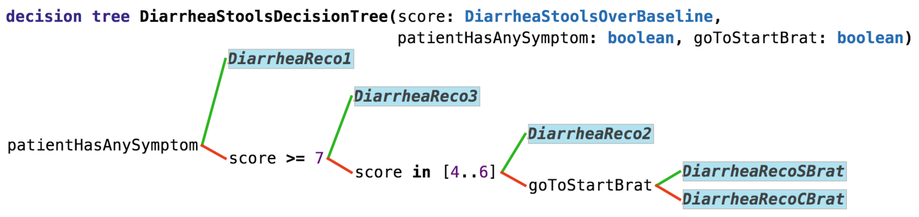
```
component DiabetesApp

val db: = createDatabase<BloodSugarData>

val missed: counter = 0

state running:
  val abs = AcquireBloodSugar.start(nextTime(), bloodSugarDB)
  on abs.ok
    abs.start(nextTime(), bloodSugarDB)
  on abs.missed
    missed.increment(1)
    abs.start(nextTime(), bloodSugarDB)
  if missed > 5
    -> error

state error:
  notifyMedicalTeam("missed blood sugar too often")
```

Decision Support. As part of the overall reactive algorithm, many complex decisions have to be made. To represent those as intuitively as possible, we have implemented a decision support language. All abstractions in that language, at a high-level, can be seen as functions: based on a list of arguments, the function returns one or more values. Plain functions are available for arithmetic calculations. However, it is typical of medical decisions that they depend on the interactions between several criteria. To improve the [READABILITY] of a function call for non-programmers, we support a style of signature that reads like a sentence fragment. For example, the function in Fig. 13 can be annotated with a syntax template that allows the following function call:

```
val riskScore = blood pressure risk for systolic <expr-1> and diastolic <expr-2>
```

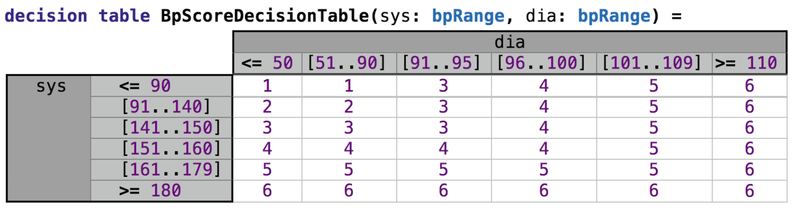
The code completion and type checks for `expr-1` and `expr-2` work as usual, but this notation provides more context for the two values a plain function call `BpScopeDecisionTable(<expr-1>, <expr-2>)`.

To improve [READABILITY] of the actual decision algorithm (and thus make it easier to validate), they are often represented as decision trees (Fig. 13) or decision tables. As mentioned in Sect. 2, basic tables and trees are available in KernelF's utility language. However, special forms are needed (and have been built specifically for this project). An example is a table that splits two values into ranges and returns a result based on these ranges. Figure 14 shows a table that returns a score; scores represent standardised severities or risks that are then used in the algorithm. KernelF's number types with ranges, and their associated static checking, is also an important ingredient to being able to improving the [ROBUSTNESS] of the algorithms.



```
decision tree DiarrheaStoolsDecisionTree(score: DiarrheaStoolsOverBaseline,
                                         patientHasAnySymptom: boolean, goToStartBrat: boolean)
```

Fig. 13. A decision tree; the green/up edges represent `yes` answers to the preceding node, the red/down edges represent `no`. (Color figure online)



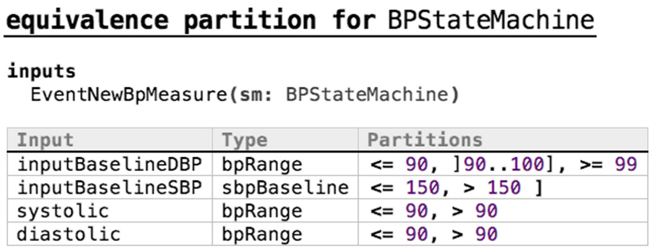
```
decision table BpScoreDecisionTable(sys: bpRange, dia: bpRange) =
```

		dia					
		<= 50	[51..90]	[91..95]	[96..100]	[101..109]	>= 110
sys	<= 90	1	1	3	4	5	6
	[91..140]	2	2	3	4	5	6
	[141..150]	3	3	3	4	5	6
	[151..160]	4	4	4	4	5	6
	[161..179]	5	5	5	5	5	6
	>= 180	6	6	6	6	6	6

Fig. 14. A decision table that specifically works on ranges of values. Note the compact syntax for range representation.

Testing. Testing is an important contributor to the success of this project, and we put significant effort into defining a suitable set of languages. For testing functions and function-like abstractions, regular JUnit-style function tests are supported; Fig. 16 shows an example. The first of the tests in Fig. 16 tests a function with one argument, the second one passes an argument list, and the last one shows how complex data structures, in this case, a patient's replies to a questionnaire, are passed to the test. The table notations for testing based on equivalence partitions in shown in Fig. 15.

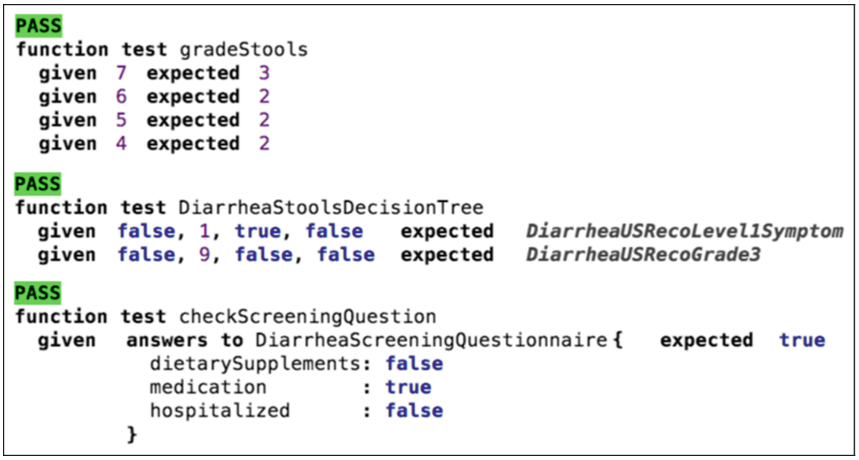
Scenario tests (Fig. 17) are more involved because they take into account the execution of the reactive main algorithm over time. They are expressed in the well-known given-when-then style,[12] which is, for example, also supported by Cucumber.[13] To express the passage of time and occurrences at specific times, the `at` notation is used. The execution of the tests is based on a simulation. The number of steps and the time resolution is derived from the scenario specification.



```
equivalence partition for BPStateMachine

inputs
  EventNewBpMeasure(sm: BPStateMachine)
```

Input	Type	Partitions
inputBaselineDBP	bpRange	<= 90,]90..100], >= 99
inputBaselineSBP	sbpBaseline	<= 150, > 150]
systolic	bpRange	<= 90, > 90
diastolic	bpRange	<= 90, > 90

Fig. 15. Equivalence partitions help test complex structures with relevant combinations of values.

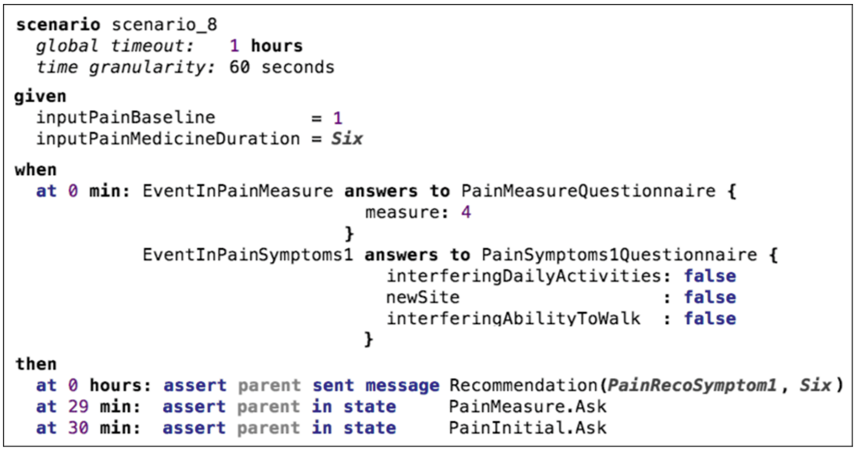


```
PASS
function test gradeStools
  given 7 expected 3
  given 6 expected 2
  given 5 expected 2
  given 4 expected 2

PASS
function test DiarrheaStoolsDecisionTree
  given false, 1, true, false   expected  DiarrheaUSRecoLevel1Symptom
  given false, 9, false, false  expected  DiarrheaUSRecoGrade3

PASS
function test checkScreeningQuestion
  given  answers to DiarrheaScreeningQuestionnaire {   expected  true
           dietarySupplements: false
           medication        : true
           hospitalized      : false
         }
```

Fig. 16. Function tests call a function (or something function-like, such as a decision tree or table) with the arguments specified after `given`, and then check that the `expected` valued is returned. The `answers` construct represents a user's reply to a questionnaire; it can be seen as an instance of a record.

Simulation. The purpose of the simulator is to let healthcare professionals "play" with an algorithm. To this end, the in-IDE interpreter executes algorithms and renders a UI that resembles the one on the phone (Fig. 18, right). A set of DSLs is available to structure the UI; lower-level styling support is available

[12] https://martinfowler.com/bliki/GivenWhenThen.html.
[13] https://cucumber.io/.



```
scenario scenario_8
  global timeout:    1 hours
  time granularity: 60 seconds
given
  inputPainBaseline          = 1
  inputPainMedicineDuration = Six
when
  at 0 min: EventInPainMeasure answers to PainMeasureQuestionnaire {
                                measure: 4
                              }
            EventInPainSymptoms1 answers to PainSymptoms1Questionnaire {
                                  interferingDailyActivities: false
                                  newSite                   : false
                                  interferingAbilityToWalk  : false
                                }
then
  at 0 hours: assert parent sent message Recommendation(PainRecoSymptom1, Six)
  at 29 min:  assert parent in state     PainMeasure.Ask
  at 30 min:  assert parent in state     PainInitial.Ask
```

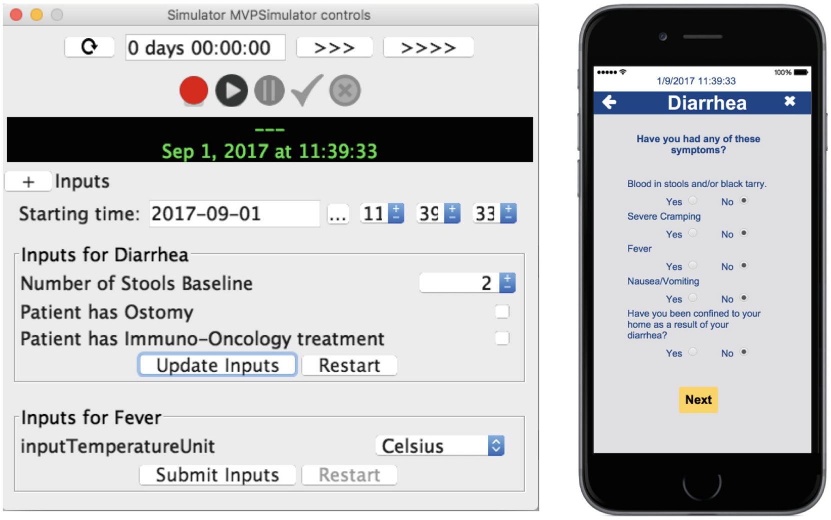
Fig. 17. Scenarios follow the established given-when-then style: *given* preconditions, *when* something happens, *then* a set of assertions must hold. Scenarios express the passage of time, as well as points in time when something happens or is asserted.

through Javascript and CSS. A control panel lets users configure a particular simulation and also fast-forward in time (Fig. 18, left). There is also a debugger that, while relying on the same interpreter, provides a lower-level view on the execution of algorithms. It is not used by HCPs.

Documentation Generation. An important output is the medical protocol, a visualisation of the complete algorithm for review by HCPs, associated medical personnel not trained in the use of the PLUTO DSLs, as well as external reviewers. The outputs are too large to show in the paper; they are essentially graphviz-style flow charts with a couple of special notational elements. It is often necessary to highlight specific aspects on the overall algorithm, so the generation of the flow chart can be configured using a DSL (Fig. 19). It supports:

- The level of detail (`Deep` in the example)
- The tags that should be included and excluded. Model elements can be tagged, for example, whether they are part of the default flow or whether they are relevant for complications in the treatment. A visualisation might highlight specific tags.
- Color mappings for tags (e.g., render the case for complications in red)
- Human-readable labels for states or messages in order to make them more understandable for outsides.

The reason why these configurations are represented as models (expressed in their own DSL) as opposed to just configuring a particular visualisation through a dialog is that many such configurations exist, and they must be reproduced in bulk, automatically, as the algorithm evolves.



Starting time: 2017-09-01 ... 11 39 33

Inputs for Diarrhea
Number of Stools Baseline 2
Patient has Ostomy
Patient has Immuno-Oncology treatment

Inputs for Fever
inputTemperatureUnit Celsius

Have you had any of these symptoms?

Blood in stools and/or black tarry.
Yes No ●
Severe Cramping
Yes No ●
Fever
Yes No ●
Nausea/Vomiting
Yes No ●
Have you been confined to your home as a result of your diarrhea?
Yes No ●

Fig. 18. Control panel to configure and execute simulations.

```
medical protocol for: ComposedStateStoolsOngoingIn24hr
  depth:     Deep
  included: everything
  excluded: #background #log
  colors
  labels:    StoolsOngoingIn24hr TITLE -> 'Diarrhea within 24 h'
             AppRecommendation -> 'Display recommendation "${message}"'
```

Fig. 19. Configuration for the generation of medical protocol flow charts.

Execution. We provide two separate execution infrastructures (Fig. 20), which is important for quality assurance, as discussed below. The first one is an in-IDE interpreter. It reuses the existing KernelF interpreter. For the functional abstractions developed in this project, we have built additional interpreters using the same interpreter infrastructure also used in KernelF. For the reactive, state-machine based part of the system, an interpreter was built using plain Java code that works on the MPS AST. It drives the overall execution and invokes the functional interpreter. A similar approach has been taken for the scenario testing DSL. The in-IDE interpreter provide short turnaround times for the users of the DSL and are an example [IDESUPPORT].

The execution on the mobile phone is based on a second interpreter. It is implemented in C++ so it can be used on iOS and Android platforms. A platform adapter provides unified access to the necessary operating system services, such as the system clock, reminders and notifications, as well as networking APIs. The C++ interpreter works on an XML representation of the AST, essentially a generic serialisation format for the MPS AST structure. Directly using the

AST is infeasible, because MPS is written in Java, and the runtime needed to be C++ for performance and portability. The reason why an interpreter was used in the first place (as opposed to generating C++ code from the algorithm) was because of the required update times: if a problem is found with the algorithm, an update has to be delivered as soon as possible. Waiting for the clearance of Apple's review team was not an option.

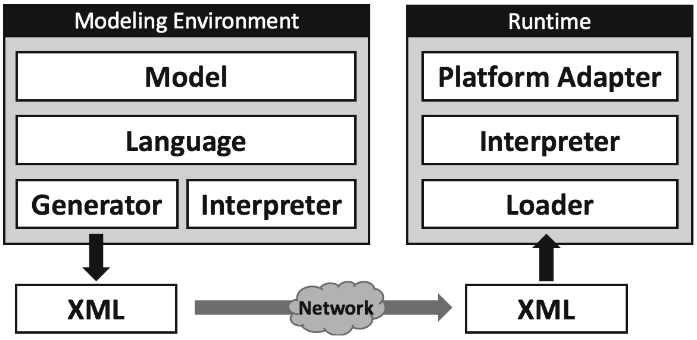
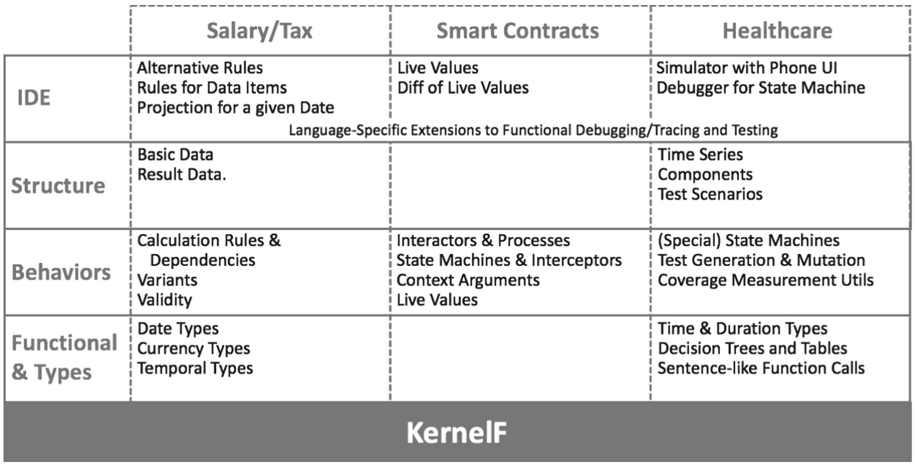


Fig. 20. Execution architecture of the languages: an IDE-interpreter plus an interpreter on the phone that works on an XML representation of the algorithms.

Quality Assurance. Ensuring the correctness of the algorithm models (validation) as well as their correct execution on the mobile phones (verification) was a major aspect of this project. Both because the well-being of human beings is directly at stake, and because the approach has to get FDA approval; otherwise the applications cannot be legally sold, jeopardising the business case. While a detailed discussion of our verification and validation approach is beyond the scope of this paper, here are the steps we took, based on a systematic risk analysis:

- Improved review-ability of the models because of the domain-oriented abstractions and notations
- Further validation of the model by healthcare experts using the simulator
- Extensive set of unit and scenario test cases that reach very high coverage of the algorithms
- Test generation to improve coverage
- Mutation testing [17] (aka fuzzing) to ensure sensitivity of tests
- Coverage measurement also of the language structure, the Java interpreter, and the C++ interpreter implementation, and 100% coverage for those
- Redundant [13] execution of all tests in the two interpreters to find random errors in each
- The two interpreters were implemented by different (teams of) developers to avoid systematic errors

– Architectural safety mechanisms such as runtime watchdogs [13] based on independently specified invariants.

~~~

This concludes our case studies. Figure 21 summarizes the extensions, aligning them with the three layers introduced at the end of Sect. 1.2. Both, the salary/tax and healthcare case studies contribute to all layers, as suggested by Sect. 1.2. The smart contracts case study is a little bit different: because it is an "experimental" set of languages, there are no domain-specific data structures or types; we used the built-in ones. Based on our experience in the logistics domain, a fully fledged contract language would need schemas, mappings to actual documents, types for money and time, as well as physical units.

As a concluding remark of this chapter, the case studies should have given the reader a good illustration of the philosophy of MPS-based language design introduced in [26]. It is really more like "libraries with syntax and type system", with lots of first-class concepts aligned closely with the application domain.



	Salary/Tax	Smart Contracts	Healthcare
IDE	Alternative Rules Rules for Data Items Projection for a given Date	Live Values Diff of Live Values	Simulator with Phone UI Debugger for State Machine
	Language-Specific Extensions to Functional Debugging/Tracing and Testing		
Structure	Basic Data Result Data.		Time Series Components Test Scenarios
Behaviors	Calculation Rules & Dependencies Variants Validity	Interactors & Processes State Machines & Interceptors Context Arguments Live Values	(Special) State Machines Test Generation & Mutation Coverage Measurement Utils
Functional & Types	Date Types Currency Types Temporal Types		Time & Duration Types Decision Trees and Tables Sentence-like Function Calls
KernelF			

Fig. 21. Overview of the extensions to functional abstractions, higher-level behavior, structures and IDE extensions for the three case studies.

6 Challenges and Open Issues

In terms of language engineering, the development of KernelF is relatively similar to the development of mbeddr, which we have evaluated extensively in [28]. This is why this paper focuses on the *language* design in the development of KernelF. However, a couple of issues are worth pointing out specifically in the context of KernelF, even though they have been mentioned generally in [28].
~~~

6.1 Type System

The type system was the biggest challenge in the current implementation. I will point out two problems that both relate to subtyping.

Number Types. The first one relates to number types. Normally, MPS determines subtype relationships via *subtyping rules*. For a given type, a subtyping rule returns the list of direct supertypes. MPS uses those to build a type hierarchy, and also uses it during type checking in situations like `val v: T = <expr>` with `expr: U`, where `U` must be a subtype of `T`. Now consider the situation where `T` is `number[0|100]` and `U` is `number[5|10]`. Clearly, the range `5..10` is a subrange of `0..100`, so the subtyping holds. But it is impossible to enumerate all supertypes of a number type, because there are infinitely many. MPS has *replacement rules* for this case. They are called as a last resort: if a type check fails, the engine tries the suitable replacement rules and sees if, by performing the specified type replacements, the type check can be made to succeed. For number types, we have defined the following replacement rule (slightly simplified):

```
replacement rule for supertype :==: NumberType as super
                 and subtype   :==: NumberType as sub {
  applicable if { sub.range.isSubrangeOf(super.range); }
  replace {}
}
```

The rule applies if two `NumberType`s are tested for a subtype relationship. It then checks if the ranges of the two types are in the required relationship. If so, the rule executes, which means the original type equation is replaced with the one given in the `replace` part. Since this is empty here, the original typing rule is effectively discarded. Since there's nothing to fail, no error is shown.

We use replacement rules for a few other reasons as well, for example, in the context of `type` definitions. Here is the catch: replacements are only executed once during the solver's attempt at solving the type system equations. So if the replacement rules create a new set of equations *which can only be solved by applying more (different) replacement rules*, this does not work. As of now, we have not found a way to solve this problem. Sprinkling explicit `cast`s over the affected programs helps, but of course this is unintuitive for the end user.

Options and Attempts. The second problem relates to the computation of supertypes in the presence of option and attempt types. Consider the following program. What is the type of `alt`?

```
fun f( ... ) = alt | <cond-1> => 42           |
                   | <cond-2> => 33.33        |
                   | <cond-3> => error(FAIL)  |
                   | <cond-3> => error(FATAL) |
```

A common supertype is typically calculated in the following way (see also [26]):

```
typing rule for AltExpression {
  var T;
  foreach alternative in node.alternatives {
    T :>=: typeof(alternative.then);
  }
  typeof(node) :==: T;
}
```

For each of the alternatives, this code submits a type equation to the solver which states that `T`, the to-be-calculated type of `alt`, is the-same-or-supertype of the type of the `then` part of the particular alternative. `T` ends up as the least common supertype of all the types of the `thens`. However, here the situation is different, the correct type is `attempt<real|FAIL, FATAL>`, i.e., the least common supertype of all non-`error` values, wrapped in an `attempt` type that lists all the possible errors. A similar issue arises if you mix values with `none`, because this introduces an `option`. Now consider the following:

```
fun f( ... ) = alt | <cond-1> => 42          |
                   | <cond-2> => 33          |
                   | <cond-3> => error(FAIL) |
                   | <cond-3> => none        |
```

There are two potentially correct types: `attempt<opt<number[33|42]>, FAIL>` and `opt<attempt<number[33|42], FAIL>>`, depending on the order of treating errors and options. We were not able to compute this type by using MPS' declarative type system DSL and resorted to imperative code. This code essentially treats attempts and options explicitly. This means, for example, that we could not implement options and attempt modularly: they are "baked into" the core type system. And one such baked in rule is that you cannot mix options and attempts; so the code above is flagged as illegal. For the DSLs we have built so far, this is an acceptable restriction.

6.2 Reactive Interpreter

Consider the following code, which might be part of a larger program (the functions) and test data (the values plus the `assertions`):

```
// test data for John
val j_last      = "Doe"
val j_first     = "John"
val j_birthYear = 1974
```

```
fun greet(f: string, l: string) = "Hello " + f + " " + l
fun age(y: int) = currentYear() - y
fun birthday(f: string, l: string, y: int) =
  "Happy " + age(y) + ". birthday, " + f + " " + l
```

```
test case Test_John {
  assert (1) greet(j_first, j_last)                 equals "Hello John Doe"
  assert (2) greet("Geddy", "Lee")                 equals "Hello Geddy Lee"
  assert (3) age(j_birthYear)                       equals 44
  assert (4) birthday(j_first, j_last, j_birthYear) equals "Happy 44. birthday, John Doe"
}
```

The Status Quo. Our current interpreter works on-demand, always runs to completion. On-demand means that a recomputation is explicitly requested. The request can happen in two ways. One way is for the user to press `Ctrl-Alt-Enter` on a program node that has a manual check (indicated through an interface implemented by the node's concept). Alternatively, the execution of manual checks (and thus, the interpreter) can be triggered by the type system, in which case MPS uses heuristics to decide when to trigger the update. In the above example (and in the current KernelF implementation), the assertions implement the required interface, so users can reevaluate an `assert` this way.

`Ctrl-Alt-Enter` also works for containers, so pressing it on the whole test case, or the surround (but not shown) tests suite recalculates all of them.

Once a recalculation is triggered it always recalculates everything, to completion. So, for example, when triggering the recomputation on the last `assert`, the interpreter for `assert` is invoked. It invokes the interpreter for the `actual` and `expected` slots. The string literal in the `expected` slot is trivial. The `actual` slot evaluates the function call. In turn, it evaluates the arguments (by calling the interpreter for the `val` references, and then, transitively, the interpreter for the init expressions on the `vals`) and then dispatches to the `birthday` function. Inside, among other things, evaluates the call to the `age` function.

Reactivity. A more scalable way would work as follows:

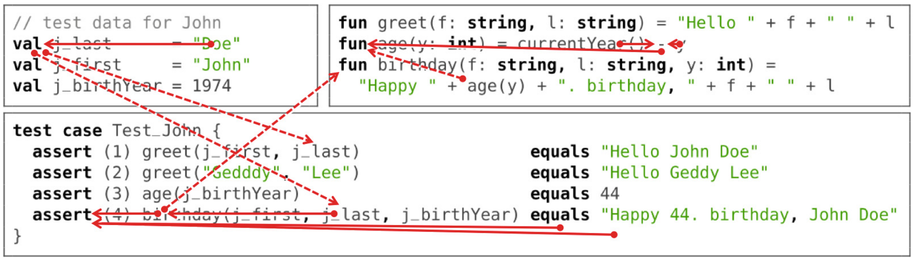
- A change to `Geddy` would trigger assertion 2
- Changing any of the `j_` values would never trigger 2
- A change to `j_last` would trigger recalculation of 1 and 4
- A change to `j_age` would trigger recalculation of 3 and 4

We would also expect that, even if 4 is recalculated because `j_last` has changed, we would *not* execute the call to `age` inside `birthday`, because the argument to `age`, `j_birthYear`, did not change. Finally, we would also expect the on-demand recalculation for changes to the program: if we change the implementation of `age`, then 3 would have to be recalculated, but also 4, because it indirectly relies on `age`. This behavior would be just like in Excel[14]: you can imagine the `vals` as cells with user-entered values, the `asserts` as cells with formulas in them and the function calls as macros. To make this reactive architecture work, the following ingredients are required:

- Change Notifications: the engine that triggers the interpreter must be notified of changes to program nodes. Since MPS is a projectional editor, and changes to the AST are already performed essentially via an architecture that relies on events, those change events are easy to get.
- Reverse Dependencies: MPS maintains a fully resolved AST, i.e., even references such as `j_first` in assertion 1 or the reference to `age` in assertion 3 are maintained as fully resolved "object pointers". However, in order to find out which parts of the program must be recomputed, the reverse dependencies are required: if the string literal `"Doe"` is changed, then we have to follow the upstream tree of containment and reference dependencies (as indicated in Fig. 22). MPS does not currently maintain (all of) these reverse dependencies. However, we assume we can maintain our own overlay data structure that is updated based on the same program change events just mentioned.
- Persistent Interpreter: Currently, the interpreter is restarted from scratch for every evaluation request (explicitly or by the type system). Restarting the interpreter means that the interpreter context, the data structure that maintains the interpreter's internal state, is also recreated, which means that all

[14] An analogy that many of our users like to draw in more ways than is good for us!

caches are empty. Thus, when a function is called with an argument for which it has been called before (and the function is pure), then the interpreter will recompute the function's result instead of reusing the one from the cache. So, again assuming a change to `"Doe"`, this triggers the recomputation of assertion 4, which calls `birthday`, which then calls `age`. Even though the argument to `age` did not change, the function is re-executed, because the (empty) cache does not know the previous result. To fix this issue, the interpreter's context (and thus, caches), would have to be maintained persistently during a user's interactive editing session.



```
// test data for John
val j_last      = "Doe"
val j_first     = "John"
val j_birthYear = 1974
```

```
fun greet(f: string, l: string) = "Hello " + f + " " + l
fun age(y: int) = currentYear() - y
fun birthday(f: string, l: string, y: int) =
  "Happy " + age(y) + ". birthday, " + f + " " + l
```

```
test case Test_John {
  assert (1) greet(j_first, j_last)                equals "Hello John Doe"
  assert (2) greet("Gedddy", "Lee")                equals "Hello Geddy Lee"
  assert (3) age(j_birthYear)                      equals 44
  assert (4) birthday(j_first, j_last, j_birthYear) equals "Happy 44. birthday, John Doe"
}
```

Fig. 22. The example code for reactive interpreters shown with the reverse dependencies relevant for a change to the value `"Doe"`. Solid lines represent containment, dashed lines represent reference dependencies.

All of these changes are absolutely feasible, and we will work on this architecture in the future. While the current implementation is not very scalable, we can, for now, live with the limitation because the in-IDE-interpreter is used for testing, and test cases are usually small and thus still run reasonably quickly. For systems that require larger integration test-style scenarios, we have explicit mocking features that act as "breakpoints" in the execution of the interpreter.

6.3 Shadow Models

Many language extension add new abstractions on top of existing ones. This means that for their semantic definition, they can be "desugared" to more basic constructs. The `alt` expression is an obvious example:

```
alt | <cond-1>   => <val-1>  |
    | <cond-2>   => <val-2>  |
    ...
    | otherwise => <val-n>   |
```

desugars to

```
if <cond-> then <val-1> else
  if <cond-2> then <val-2> else
    ...
    else <val-n>
```

It is idiomatic for MPS generators to be stacked, and they can be scheduled to perform desugarings to a base language, before that language is processed further. Essentially, all of mbeddr's C extensions are translated this way. It would be nice if the same approach could be used with interpreters as well: programs are reduced to their most basic form, which is then submitted to the interpreter. This way, the interpreter only has to be defined for a minimal language. More importantly, the same desugaring could be used independent of what is done with the desugared, basic form of the program: it could be interpreted, submitted to a Java generator, or translated to the solver. You can see while this approach is very desirable for reasons of reduced effort and improved quality.

The reason why the approach works well with generators is that those are executed on demand; when the user requests a (re-)build of the model, the cascade of generators is executed according to their relative priorities ("higher" desugarings first). However, the interpreter is expected to run interactively, which means, very fast: as the user changes parts of the program, the interpreter should be executed and the results updated. The same is true for the checks performed with the solver. What we would need is an incremental maintenance of the desugared (or otherwise derived) models. While it is easy in MPS to receive fine-grained notification of changes to programs, we have not yet found a way of expressing the necessary incremental graph transformations. While we are actively working on this challenge, for now, every language concept requires a native interpreter, i.e., one that is specifically implemented for the (potentially desugarable) language concept.

7 Related Work

7.1 Dynamic Languages

A widespread approach for building embedded DSLs is the use of dynamic languages that support reflection and flexible syntax. Prime examples are Groovy and Ruby. However, the approach is not suitable for our purposes, for several reasons. First, the implementation based on reflection prevents static analysis and (automatic) IDE support. Second, the syntax of extensions is limited to the freedom given by the grammars of the respective language.[15] In addition, the languages are all not purely functional and provide no support for explicit effects tracking. We discarded this option early and clearly.

7.2 Other Base Languages

mbeddr C. mbeddr [30] is an implementation of C in MPS. It uses the same extension mechanisms as KernelF because it is built on MPS as well. Like KernelF, mbeddr C is implemented in a modular way, i.e., even the core of C is split into several languages. One of them, `com.mbeddr.core.expressions`, contains only the C expressions and primitive types. In particular, it does not have user-defined data types, pointers, statements, or a module system. The idea was to

[15] Both of these points are clearly illustrated by a customer's (not very satisfying) attempt at building a whole range of business DSLs with Groovy.

make this a kind of core expression language to be hosted in other DSL. In practice, this works well *as long as that DSL generates to C.* However, even in this core language subset, there are many implicit assumptions about C, making it unsuitable as a generic, embeddable expression language; building an interpreter is also tough. It also misses many useful features, such as higher-order functions.

When we started seeing the need for a core expression language, we thought about generalising the mbeddr expressions; however, we decided against it and started KernelF: the required changes would have been too great, making mbeddr C too complicated. The use cases are just too different.

MPS BaseLanguage. MPS ships with a language called BaseLanguage – it wears its purpose clearly on its sleeve. It is fundamentally a slightly extended version of Java (for example, it had higher order functions and closures long before they were standardised as part of Java 8). It also ships with a set of (modular) extensions for meta programming, supplying language constructs, to, for example, create, navigate and query ASTs.

BaseLanguage has been used successfully – by us and others – as the basis for DSLs. If those DSLs either extend Java or at least generate to Java, BaseLanguage is a great fit and the recommended way to go. Even though it is not built in a modular way, MPS' support for restricting languages using constraints is powerful enough to cut it down to what is relevant in any particular DSL.

However, similar to mbeddr C, it suffers from its tight connection to Java in terms of data types, operators and assumptions about the context in which expressions are used. The fact that it is not a purely functional language and does not support effects tracking also makes it much harder to analyze. It also has several features, such as generics, that make it harder to extend. Finally, its long evolution in MPS also means that it carries around a lot of baggage; we decided that it is worth the effort to build a new, clean base language.

Xbase/Xtend. Xbase [8] is a functional language that ships with Xtext[16]. Similar to KernelF, its purpose is to be extended and embedded in the context of DSLs. Xtend[17] is a full programming language (with classes, modules and effects) that embeds Xbase expressions. Similar to Kotlin[18] and Ceylon[19], its goal is to be a better, cleaned up Java, while not being as sophisticated/complex as Scala. For the purposes of being an embeddable base language, Xtend's scope is too big (like Java or C), so we limit our discussion in this paragraph to Xbase.

In terms of its suitability as a base language, Xbase suffers from several problems. The most obvious one for our use case is that it is implemented in Xtext, and is thus useless for MPS-based languages. Of course, this does not say anything about its conceptual suitability as a core language. However, there are also two significant conceptual problems. First, because of the fact that it is implemented in Xtext, its support for modular extension or embedding are

[16] https://www.eclipse.org/Xtext/.
[17] http://www.eclipse.org/xtend/.
[18] https://kotlinlang.org/.
[19] https://ceylon-lang.org/.

limited: one cannot use several independently developed extensions in the same program in a modular way. Consequently, no such extensions are known to us, or documented in the literature. Second, Xbase is very tightly coupled to Java: it uses Java classes, generates to Java and even its IDE support is realized by maintaining Java shadow models in the background. While this is a great benefit for Java-based languages (the goal of Xbase), it is a drawback in general.

In terms of its core abstractions, many of the ideas in KernelF and Xbase are similar: everything is an expression, functional abstractions, no modules or statements (those are supplied by Xtend).

7.3 Lisp-Style Languages

Lisp-style languages have a long tradition of being extensible with new constructs and being used at the core of other systems, such as Emacs. Racket[20] takes this to an extreme and allows significant syntactical flexibility for Lisp or extensions. We decided against this style of language for several reasons:

First, while, generally, it is a matter of taste (and of getting used to it) whether developers like or hate the syntax, it is very clear that (our) end users do not like it. Thus, adopting this syntactical style was out of the question.

Second, existing Lisp implementations are parser-based, and even the meta-programming facilities rely on integrated parsing through macros. This limits the syntactic freedom to textual notations in general, and to the capabilities of the macro system more specifically. We needed more flexibility.

Third, we wanted language extensions to be first-class: instead of defining them through meta programming, we wanted the power of a language workbench. Of course we could have implemented (a version of) Lisp im MPS and then used MPS' extension mechanisms to build first-class extensions. However, then we would not make use of Lisp's inherent extensibility, while still getting the end-user-unsuitable syntactic style – clearly not a good tradeoff.

Finally, Lisp language extensions only extend the *language*, not the IDE. However, for our use cases, the IDE is just as important as the language itself, so any language extension or embedding must also be known to the IDE. Lisp does not support this (at least not out of the box).

7.4 Embeddable Languages

Lua[21] is a small and embeddable language. In contrast to KernelF, it is not functional – it has effects and statements. Also, the notion of extension relates to extending the C-based runtime system, not the front-end syntax. So, out of the box, Lua would not have been an alternative to the development of KernelF.

However, we could have reimplemented Lua in MPS and used MPS' language engineering facilities for syntactic extension. While possible, this would still mean that we would use a procedural language as opposed to a functional one, which

[20] https://racket-lang.org/.
[21] https://www.lua.org/.

was at odds with our design goals. On the plus side is Lua's small and efficient runtime system. While we did not perform any comparisons, it is certainly faster than our MPS-integrated AST interpreter. However, performance considerations are not a core requirement for the IDE-integrated interpreter. If fast execution is required, we generate to Java or C, or implement reactivity (Sect. 6.2).

7.5 Other Language Workbenches

This paper is not about evaluating MPS' suitability as a language workbench; see [28] instead. Thus, a detailed evaluation about alternative implementation technologies for KernelF is outside the scope of this paper. Nonetheless, if, for some reason, we could not use MPS for KernelF and our customer projects, Racket would probably be the best alternative.

8 Conclusion

We have built KernelF as a base language for DSLs. This means that it must be extensible (so new, domain-specific language constructs can be added), embeddable (so it can be used as part of a variety of host languages) and language concepts users do not need must be removable or replaceable. Our case studies show that we have achieved this goal. Since developing KernelF, we have used it in most customer projects that required expressions or a full-blown programming language as a basis.

Why were we successful? Two factors contribute. One is that we have built KernelF after years and years of building DSLs. So we had a pretty good understanding of the features required for the language, and to make it extensible and embeddable. In particular, the design that enables extensibility was based on our experience with mbeddr C, which has proven to be extensible as well. We also had a good understanding of what features *not* to include, because they are typically contributed by the hosting DSL. The second factor is MPS itself. As we have analyzed in [28], MPS supports this kind of modular language engineering extremely well.

We continue to use KernelF as a basis for our DSL work. We are also using it as the core of a set of meta languages in our new web-based language workbench Convecton. Once it is expressive enough, we will implement KernelF in Convecton so we have it available as a base language for Convecton-based DSLs as well.

~~~

Acknowledgements. I implemented most of KernelF myself. However, this would not have been possible without the team at itemis: they were sparring partners in design discussions, they helped mature the language by using and stressing it, they built some of the features in the case studies, and generally provided the fertile ground on which something like KernelF can flourish. I also want to thank our customers. Not just those of the particular systems described in the case studies, but all of them:
~~~

without their trust in us and, ultimately, their money, none of what is discussed in this paper would have happened. Finally, I want to thank the MPS team at Jetbrains for building an amazing tool and for helping us use it productively over the years.

References

1. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, 8–9 January 2018, Los Angeles, CA, USA, pp. 66–77 (2018). https://doi.org/10.1145/3167084
2. Berger, T., Völter, M., Jensen, H.P., Dangprasert, T., Siegmund, J.: Efficiency of projectional editing: a controlled experiment. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 763–774. ACM (2016)
3. Booth, S.P., Jones, S.B.: Walk backwards to happiness: debugging by time travel. In: Proceedings of the 3rd International Workshop on Automatic Debugging (AADEBUG 1997), no. 001, pp. 171–184. Linköping University Electronic Press (1997)
4. Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., Combemale, B.: Execution framework of the GEMOC studio (tool demo). In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, pp. 84–89. ACM (2016)
5. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
6. DeRemer, F., Kron, H.H.: Programming-in-the-large versus programming-in-the-small. IEEE Trans. Softw. Eng. **2**, 80–86 (1976)
7. Douceur, J.R.: The sybil attack. In: Druschel, P., Kaashoek, F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 251–260. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45748-8_24
8. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for Java. ACM SIGPLAN Not. **48**, 112–121 (2012)
9. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, p. 7. ACM (2012)
10. Fowler, M.: Language workbenches: the killer-app for domain specific languages (2005)
11. Frantz, C.K., Nowostawski, M.: From institutions to code: towards automated generation of smart contracts. In: IEEE International Workshops on Foundations and Applications of Self* Systems, pp. 210–215. IEEE (2016)
12. Gibbons, R.: A Primer in Game Theory. Harvester Wheatsheaf, Bushey (1992)
13. Hanmer, R.: Patterns for Fault Tolerant Software. Wiley, Chichester (2013)
14. Hickey, R.: The Clojure programming language. In: Proceedings of the 2008 Symposium on Dynamic Languages, p. 1. ACM (2008)
15. Hirai, Y.: Defining the Ethereum virtual machine for interactive theorem provers. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
16. Jensen, C.S., Snodgrass, R.T.: Temporal data management. IEEE Trans. Knowl. Data Eng. **11**(1), 36–44 (1999)

17. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
18. Korpela, K., Hallikas, J., Dahlberg, T.: Digital supply chain transformation toward blockchain integration. In: Proceedings of the 50th Hawaii International Conference on System Sciences (2017)
19. Narasimban, P., Moser, L.E., Melliar-Smith, P.M.: Using interceptors to enhance CORBA. Computer **32**(7), 62–68 (1999)
20. Odersky, M., Altherr, P. Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the Scala programming language. Technical report (2004)
21. Peters, G.W., Panayi, E.: Understanding modern banking ledgers through blockchain technologies: future of transaction processing and smart contracts on the internet of money. In: Tasca, P., Aste, T., Pelizzon, L., Perony, N. (eds.) Banking Beyond Banks and Money, pp. 239–278. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42448-4_13
22. Tapscott, D., Tapscott, A.: Blockchain Revolution: How the Technology Behind Bitcoin is Changing Money, Business, and the World. Penguin, Toronto (2016)
23. Voelter, M.: A smart contract development stack. Posted 6 December 2017
24. Voelter, M.: Language and IDE modularization and composition with MPS. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2011. LNCS, vol. 7680, pp. 383–430. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35992-7_11
25. Voelter, M.: The kernelF reference (2018)
26. Voelter, M.: Language development with MPS - a quick overview (2018)
27. Voelter, M., vand Deursen, A., Kolb, B., Eberle, S.: Using C language extensions for developing embedded software: a case study. In: Proceedings of OOPSLA 2015, pp. 655–674. ACM (2015)
28. Voelter, M., Kolb, B., Szabó, T., Ratiu, D., van Deursen, A.: Lessons learned from developing mbeddr: a case study in language engineering with MPS. Softw. Syst. Model. (2017). https://doi.org/10.1007/s10270-016-0575-4
29. Voelter, M., Lisson, S.: Supporting diverse notations in MPS' projectional editor. In: GEMOC Workshop (2014)
30. Voelter, M., Ratiu, D., Kolb, B., Schaetz, B.: mbeddr: instantiating a language workbench in the embedded software domain. Autom. Softw. Eng. **20**(3), 1–52 (2013)
31. Voelter, M. Szabó, T., Lisson, S., Kolb, B., Erdweg, S., Berger, T.: Efficient development of consistent projectional editors using grammar cells. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, pp. 28–40. ACM (2016)
32. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Pap. **151**, 1–32 (2014)

Full Papers

Virtual Network Embedding: Reducing the Search Space by Model Transformation Techniques

Stefan Tomaszek[1(✉)], Erhan Leblebici[1], Lin Wang[2], and Andy Schürr[1]

[1] Real-Time Systems Lab, TU Darmstadt, Darmstadt, Germany
{stefan.tomaszek,erhan.leblebici,andy.schuerr}@es.tu-darmstadt.de
[2] Telecooperation Lab, TU Darmstadt, Darmstadt, Germany
wang@tk.tu-darmstadt.de

Abstract. Virtualization is a promising technology to enhance the scalability and utilization of data centers for managing, developing, and operating network functions. Furthermore, it allows to flexibly place and execute virtual networks and machines on physical hardware. The problem of mapping a virtual network to physical resources, however, is known to be $\mathcal{NP}$-hard and is often tackled by optimization techniques, e.g., by (ILP). On the one hand, highly tailored approaches based on heuristics significantly reduce the search space of the problem for specific environments and constraints, which, however, are difficult to transfer to other scenarios. On the other hand, ILP-based solutions are highly customizable and correct by construction with a huge search space. To mitigate search space problems while still guaranteeing correctness, we propose a combination of model transformation and ILP techniques. This combination is highly customizable and extensible in order to support multiple network domains, environments, and constraints allowing for rapid prototyping in different settings of virtualization tasks. Our experimental evaluation, finally, confirms that model transformation reduces the size of the optimization problem significantly and consequently the required runtime while still retaining the quality of mappings.

Keywords: Virtual network embedding · Integer linear programming
Model-driven development · Triple graph grammar · Data center

1 Introduction

Online services such as social networking, e-commerce, and other web applications are ubiquitous today and place high demands on service providers in terms of availability and scalability. The huge amount of data generated during analysis and processing pushes traditional network topologies and administrations to their limits. In order to meet the high requirements for availability and scalability as well as to realize fast development cycles, cloud computing has emerged as the leading technology in this area. Data centers form the central facility for cloud computing to provide the large number of computing and storage servers. For operating these highly complex environments, hardware virtualization has established itself as a viable technology for decoupling the underlying infrastructure and applications. In this regard, virtualized environments can be deployed in a very flexible, scalable, and cost-effective manner. Encapsulating services in virtual machines and (VNs), which are decoupled as logical units from the physical hardware, increases the flexibility to deploy them automatically via software and to achieve a high utilization of multiplexed hardware resources. These virtualization techniques make it possible to standardize the configuration, reduce energy consumption, act independently of hardware, and enable a fast development process.

The advantages of VNs, however, are accompanied by a high degree of complexity, which is particularly evident in the problem of virtual network embedding (VNE). This problem describes the embedding of VNs into a substrate network (SN), e.g., into a data center, whereby restrictions on multiple dimensions have to be observed and certain optimization goals such as maximization of resource utilization have to be fulfilled. These restrictions apply to both server and link properties including memory, computation capacity, and bandwidth as well as to functional and non-functional requirements such as security zones or service level agreements. The abundance of possibilities in combination with different network topologies, application scenarios, and optimization goals makes it difficult to adapt, develop, and simulate embedding algorithms.

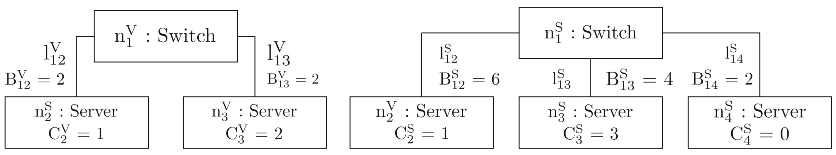
Due to the increasing importance of cloud computing and network virtualization, research into the automation of VNE has been intensified. Since VNE is known to be an $\mathcal{NP}$-hard optimization problem with a considerable search space [1], a variety of algorithms have been developed to significantly reduce the search space (and consequently the required runtime) for this problem. In particular, the following two groups of approaches seem to be promising (we refer to [1,2] for a survey): Heuristics-based, e.g., [3–5], and integer linear programming (ILP)-based approaches, e.g. [5,6]. In the first case, practical yet case-specific methods tailored for (a family of) certain infrastructures and application scenarios can massively reduce the search space to obtain an approximately good result for the VNE problem even in large data centers. For example, Guo et al. [4] introduces a heuristics-based approach to map virtual networks in the smallest possible subset of a tree-based data center by respecting bandwidth constraints. Zeng et al. [5] additionally consider the traffic between virtual machines and try to minimize the total communication cost among them. However, no guarantees can be given regarding adherence to all basic constraints, and the adaption of these approaches to other environments and network topologies is difficult if not impossible. In the second case, more generalized approaches based on ILP support a wide range of applications, whereby compliance with the constraints and requirements are ensured and an optimal result is achieved. However, these approaches suffer from the large search space of the formulated optimization problem limiting their applicability to smaller data centers [6].

To close the gap between highly adapted heuristics and purely ILP-based solutions, we propose a (MdVNE) approach for capturing VNE problems as a combination of model transformation (MT) and ILP techniques. In this setting, MT rules define which situations are to be considered as allowed mappings at all between individual VN and SN elements and, consequently, the result of an MT run is a set of potential mappings. Subsequently, ILP techniques are used to reach an ultimate decision for the collected potential mappings. This way, MT is utilized to reduce the search space among all possibilities of mappings, whereas ILP operates in this reduced search space yet still provides its strengths with regard to correctness and optimality. From an implementation point of view, our concept for the combination of MT and ILP allows for developing new and customized VNE algorithms at a higher level of abstraction (achieved with MT) while still respecting the optimization characteristics of the VNE problem.

Based on our first ideas towards combining MT and ILP for VNE presented in [7], our contributions in this paper are threefold:

(i) We provide a set-theoretical identification of search spaces involved in VNE when tackled by MT, ILP, or a combination of both. This is useful to conceptually locate the advantage of combining MT and ILP as compared to using purely MT or ILP.
(ii) We informally demonstrate how to use different MT rules to reduce the search space of potential mappings in VNE differently and flexibly (depending on, e.g., available heuristics or case-specific concerns that can be incorporated into the MT rules).
(iii) Based on our concrete implementations, we experimentally evaluate how the conceptually expected advantage of combining MT and ILP is reflected in practice in terms of required runtime as well as the size of the formulated optimization problem.

Throughout the paper the network topology from Fig. 1 is used as a running example to explain the VNE problem. On the virtual side, the VN consists of one central switch (n_1^V) and two servers (n_2^V, n_3^V), as well as bidirectional links (l_{12}^V, l_{13}^V) from each server to the switch. On the substrate side (representing the physical network), the SN is similar to the VN with one central switch and three servers with different computing capacities and links from each server to the switch. Note that the superscripts V and S in our notation refer to the virtual and substrate networks, respectively.



(a) Virtual Network (b) Substrate Network

Fig. 1. Running example with a virtual and substrate network.

The structure of the paper is organized as follows. Section 2 describes the VNE problem as an ILP-based problem description. Afterwards, Sect. 3 presents the MdVNE approach with its different search spaces. Following an evaluation in Sects. 4 and 5 presents related work. Finally, a summary and a brief discussion of future work concludes the paper.

2 ILP-Based Problem Description

In this section we introduce and define the VNE problem as an ILP formulation to achieve the best solution for a linear optimization function subject to a set of linear equalities and inequalities as constraints [8]. The definition of constraints and objectives in the following serves to give an overview of state-of-the-art solutions based on ILP (before incorporating MT in the next section) and is inspired by Sahhaf et al. [9].

2.1 Substrate Model

The SN is given as an undirected graph $G^S = (N^S, L^S)$ containing a set of substrate nodes N^S and substrate links L^S. In addition, there are paths $p_{uv} \in P^S$ consisting of acyclic connected links with the source node u and target node v.

$$P^S = \{p | p \subseteq L^S \text{ and the links in } p \text{ lead to an acyclic path}\}$$

Each substrate node and link has a certain amount of resources which can be used by any virtual network. In this paper computing capacity (C), memory (M), storage (S), and bandwidth (B) are considered.

$$\forall u \in N^S : C_u^S, M_u^S, S_u^S \in \mathbb{N}^+; \forall e \in L^S : B_e^S \in \mathbb{N}^+$$

Additionally, every substrate node has a set of service types (Sr, Sw) to define which services (Server, Switch) may run on this node.

$$\forall u \in N^S : \begin{cases} u^{Sr}, u^{Sw} \in \{0,1\} \\ u^{Sr} = 1 \text{ iff } u \text{ may host a Server} \\ u^{Sw} = 1 \text{ iff } u \text{ may host a Switch} \end{cases}$$

2.2 Virtual Model

We model the VN similar to the SN as $G^V = (N^V, L^V)$ with the virtual nodes N^V and the virtual links $e_{ij} \in L^V$ with source node i and target node j. The resources and services are similarly defined as those of the SN except that every virtual node implements exactly one service.

$$\forall i \in N^V : \begin{cases} i^{Sr}, i^{Sw} \in \{0,1\}, i^{Sr} + i^{Sw} = 1 \\ i^{Sr} = 1 \text{ iff } i \text{ may host a Server} \\ i^{Sw} = 1 \text{ iff } i \text{ may host a Switch} \end{cases}$$

2.3 Mapping Variables

For the placement of the nodes and links we define two sets of mapping variables. The variable x_u^i is used to indicate if the virtual node i is mapped to a substrate node u or not.

$$\forall i \in N^V, \forall u \in N^S, x_u^i \in \{0, 1\}$$

The second set of mapping variables $f_{p_{uv}}^{e_{ij}}$ is used to assign the placement of a virtual link e_{ij} to one substrate path p_{uv}. Therefore, if $f_{p_{uv}}^{e_{ij}} = 1$ the virtual link e_{ij} is mapped to the substrate path p_{uv}.

$$\forall e_{ij} \in L^V, \forall p_{uv} \in P^S, f_{p_{uv}}^{e_{ij}} \in \{0, 1\}$$

2.4 Constraints

To ensure that all technical, functional and non-functional requirements of the mapping from VNs to an SN are met, additional constraints are needed. These constraints can be divided into node and link constraints which have restrictions on the node or link level, respectively. A mapping can only be deployed if all constraints are fulfilled. If no mappings fulfilling these constraints are found, the embedding request is rejected.

Node Constraints. Node constraints ensure that the demands of the virtual nodes are satisfied, supported service types of the substrate nodes are respected, and resources of the substrate nodes are not overbooked. The first constraint (1) ensures that every virtual node is mapped to exactly one substrate node. Given $N^S = \{u_1, ..., u_n\}$, $\sum x_u^i$ denotes the sum $x_1^i, ..., x_n^i$.

$$\forall i \in N^V : \sum_{u \in N^S} x_u^i = 1 \tag{1}$$

In the second constraint (2), a substrate node u must be able to host the service types of all virtual nodes mapped to u. We use $\leq$ in the following constraints as a logical implication. For example, choosing a mapping x_u^i while $i^{Sr} = 1$ implies that $u^{Sr} = 1$, i.e., $i^{Sr} x_u^i \leq u^{Sr}$ (note that i^{Sr} is a constant here given by the VN and equals to 1 iff the virtual node i hosts a server, while x_u^i is further on the mapping and thus the decision variable).

$$\forall i \in N^V, \forall u \in N^S : i^{Sr} x_u^i \leq u^{Sr}, i^{Sw} x_u^i \leq u^{Sw} \tag{2}$$

The last node constraint (3) ensures that the resources of a substrate node u are not overbooked by the demands of the virtual nodes mapped to u. The used resources, e.g., computing capacity C, are coefficients for a mapping variable x_u^i and flow into the sum iff $x_u^i = 1$.

$$\forall u \in N^S : \sum_{i \in N^V} C_i x_u^i \leq C_u, \sum_{i \in N^V} M_i x_u^i \leq M_u, \sum_{i \in N^V} S_i x_u^i \leq S_u \tag{3}$$

Link Constraints. The constraints for the links are similar to the constraints for the nodes. Constraint (4) ensures that every virtual link is mapped to one substrate path and that the source/target node of the virtual link is also mapped to the source/target node of the substrate path. We again use implications ($\leq$) for the latter.

$$\begin{aligned} \forall e_{ij} \in L^V : \sum_{p_{uv} \in P^S} f^{e_{ij}}_{p_{uv}} = 1 \\ \forall e_{ij} \in L^V, \forall p_{uv} \in P^S : f^{e_{ij}}_{p_{uv}} \leq x^i_u \text{ and } f^{e_{ij}}_{p_{uv}} \leq x^j_v \end{aligned} \tag{4}$$

The last constraint (5) guarantees that the resources of a substrate link e is not overbooked by the virtual links mapped to a path containing e. Again, the used resources (bandwidth B) are coefficients for mapping variables and flow into the sum iff the respective mapping is chosen.

$$\forall e \in L^S : \sum_{e_{ij} \in L^V} \sum_{\substack{p_{uv} \in P^S, \\ e \in p_{uv}}} B_{e_{ij}} f^{e_{ij}}_{p_{uv}} \leq B_e \tag{5}$$

2.5 Objective Function

The objective function to solve the VNE problem can be different in every scenario or for every service provider. Throughout the paper the objective to minimize the costs for the embedding of the VN to the SN is used. Therefore, a cost function is used for each mapping of a virtual to a substrate element. While the functions $cost^N$ and $cost^L$ below abstract the costs for mappings in our formalization, possible reference points for a cost calculation in practice include required resources of virtual elements, hardware properties of substrate elements, and hosted service types.

$$cost^N : (N^S \times N^V) \rightarrow \mathbb{R}^+, cost^L : (P^S \times L^V) \rightarrow \mathbb{R}^+$$

Finally, the following objective function minimizes the costs for mappings.

$$\text{min:} \sum_{u \in N^S} \sum_{i \in N^V} x^i_u cost^N(u,i) + \sum_{p_{uv} \in P^S} \sum_{e_{ij} \in L^V} f^{e_{ij}}_{p_{uv}} cost^L(p_{uv}, e_{ij})$$

2.6 Search Spaces

To clarify the search spaces and the solution space for the ILP-based approach, the Venn diagram from Fig. 2 is used. In this diagram, *VNE* describes the entire search space for this problem. The specification in an ILP-based approach restricts the search space to the *ILP* area, whereby the search space is only reduced by potential solutions that do not meet the requirements. Since we assume that the ILP-based approach meets all constraints, is correct by construction, and finds the optimal solution

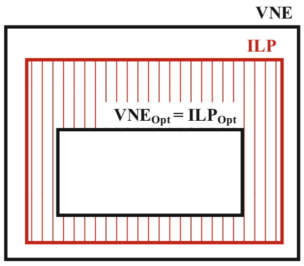


Fig. 2. Venn diagram for an ILP-based approach.

if a result exists, the optimal solution space for the VNE problem (VNE_{Opt}) is identical to the solution space of the ILP-based approach (ILP_{Opt}).

3 MdVNE Approach

In our previous work [7], the MdVNE approach, combining model-driven development and ILP technologies, is introduced for a restricted set of resources and a fixed optimization goal. We extend MdVNE in the following for multiple resources, demands, services, and optimization goals. Therefore, the meta model is redesigned to support multiple nested networks, embedding scenarios, and is easily extensible to other network domains like wireless or telecommunication networks. Also, user specific constraints can be added and the optimization goal is adjustable using the weighted-sum method. Additionally, we provide a set-theoretical identification of search spaces involved in this approach with the aim to demonstrate the reduction of the search spaces by using MT-based techniques.

3.1 MdVNE Process

A schematic view of the MdVNE mapping process is shown in Fig. 3. In step (1) a user specifies VN requests (VNRs) for an existing SN already hosting some VNs. The green + indicates that a new VN request is defined and waiting to be deployed. In step (2), all possible mappings respecting the transformation constraints (e.g. structural constraints) are generated by using MT technologies to reduce the search space resulting in a set of different possible mapping candidates (step (3)). After that, these mapping options are encoded as an ILP problem (step (4)) with additional constraints not covered by the MT rules. Then the ILP solver returns an optimal solution for the given search space if one exists. This solution is deployed in step (5). After that, the VN requests are hosted in the SN and the next VN request can be mapped.

The MT in Fig. 3 is defined in the form of rules describing how to map a source onto a target model based on the meta model which specifies the family of models that can be expressed. Our MT rules are inspired by so-called (TGGs) [10] which construct two graphs (VN and SN) together with correspondences (mapping relationship between virtual and substrate networks in our concrete case). ILP is then used to solve the optimization problem expressed as linear inequalities. Transforming the mapping candidates generated using MT into an ILP problem with added global and user specific constraints is described in detail in [7].

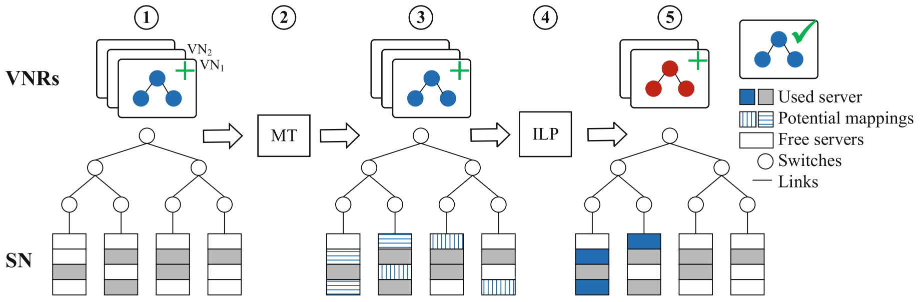


Fig. 3. Schematic view of the MdVNE mapping process.

3.2 Search Spaces

To clarify the search spaces for MdVNE, the Venn diagram from Fig. 2 has been expanded and merged in Fig. 4. *ILP* further on describes the search space of a purely ILP-based approach. In addition, the search space *MT* and the solution space *MdVNE* have been added to this diagram. *MT* describes the search space that is stretched by the concrete MT rules and is located within the *ILP* search space since the amount of mapping candidates is in the worst case comparable to an ILP specification due to the underlying meta model and the MT language. Moreover, we denote a translation $\mathcal{T} : MT \rightarrow ILP$ that maps the MT result to an ILP problem. The output of $\mathcal{T}$, i.e., ILP constraints and an objective function, is formed in line with our formalization in the previous section (we refer to [7] for how to realize such a translation, while the form of ILP constraints and objectives as provided in the previous section suffices to understand the current discussion). Most importantly, the variables in the formulated ILP problem are reduced to the set of potential mappings collected by MT instead of a pairwise consideration of all possible mappings. This way, certain mapping conditions (in particular structural ones) are already respected by the MT run, whereas further conditions that go beyond the MT capabilities and the optimization goal are again tackled with ILP.

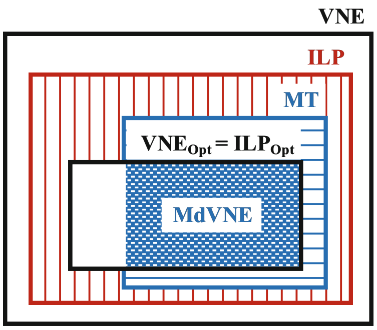


Fig. 4. Venn diagram for the MdVNE and ILP-based approach.

Overall, we distinguish between the following search spaces in Fig. 4:

– The search space *VNE* describes the set of all expressible mappings of a VN to a SN.
– *ILP* and *MT* denote the set of all expressible mappings in the technical domains of ILP and MT, respectively.

- VNE_{Opt} denotes the set of all optimal mappings of the VNE problem respecting all constraints which is identical to the solution space of ILP_{Opt}.
- The intersection $MdVNE$ of VNE_{Opt} and MT is the solution space for MdVNE respecting all constraints of the VNE problem.

With regard to the translation $\mathcal{T}$, we distinguish between the following cases:

(i) $MT = ILP$: MT techniques do not reduce the search space resulting in a purely ILP-based approach.
(ii) $|MT| \leq 1$: This results in a purely MT-based approach.
(iii) $MT \cap VNE_{Opt} \neq \emptyset$: This results in a mixed ILP and MT-based approach that finds an optimal solution for the problem.
(iv) $MT \cap VNE_{Opt} = \emptyset$: This results in a heuristics-based approach without finding an optimal solution.

Among the different cases stated above, our goal and demonstrations in the rest of the paper focus on (iii), solving the $\mathcal{NP}$-hard VNE problem [1,2] still via ILP but with a significant reduction of the search space and thus a strong influence on the required runtime.

3.3 Example

The example introduced in Fig. 1 is used to illustrate the possibilities to reduce the search space by using MT-based technologies in comparison to a purely ILP-based approach. In Fig. 5, TGG rules are specified which restrict the search area to different degrees. Note that these rules are used to calculate all possible mappings, which are then passed as input to the ILP solver. Each TGG rule consists of elements from VN (left side) and SN (right side) with the mapping elements in between. Green elements marked with ++ are created by the rule, whereby the VN is element-wise mapped to the SN. In rule (a), all virtual servers are potentially mapped to substrate servers, whereby the restriction here is only the type of each element. In rule (b), potential mappings are restricted such that virtual servers are only mapped to the substrate servers if the switches connected to the servers have already been mapped to each other and the computing capacity of the virtual server is less than or equal to the computing capacity of the substrate server. The last rule (c) can be especially considered as a heuristics-based approach, since the entire VN is mapped to a single substrate server, which directly leads to a solution. While doing so the sum of the computing capacities of all virtual servers must not exceed the computing capacity of the substrate server.

After defining the MT rules, we discuss the mapping candidates created for the concrete instances from Fig. 1 when applying these rules. Figure 6 shows the mapping candidates as dashed lines between the nodes once for the purely ILP-based approach and for the MT-based approach with two different rule sets. In the ILP-based approach, Fig. 6 shows all combinatorially possible mappings of nodes that represent the mapping variable x_u^i and thus the ILP search space. Figure 6 (1) uses the two rules (a) and (b) from Fig. 5 for the MT-based approach, disregarding the attribute condition $C_1^V \leq C_1^S$. In case (2) in Fig. 6 the

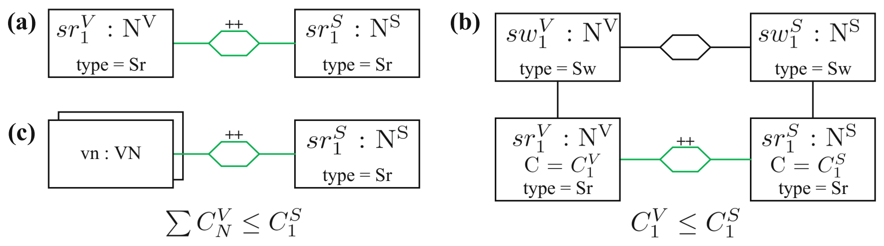


Fig. 5. Example for different TGG rules.

attribute condition $C_1^V \leq C_1^S$ is also used, resulting in fewer mapping candidates. It becomes clear that the *MT* search spaces from MdVNE in Fig. 6 can be strongly restricted in this example by the rules and their attribute conditions. That is, the style of the MT rules related to structural and/or attribute conditions has a decisive impact of the resulting search space as demonstrated here.

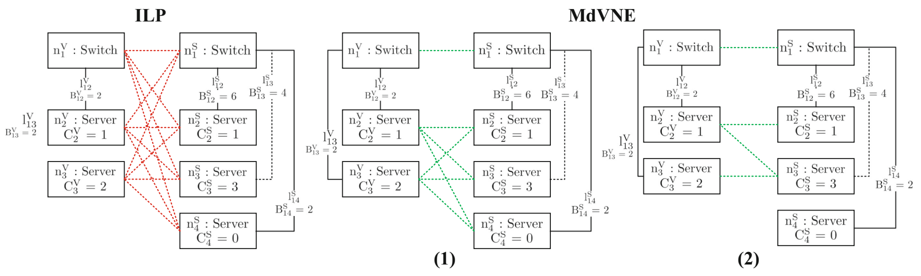


Fig. 6. Example for potential mappings in the *ILP* and *MT* search spaces.

4 Evaluation

In this section, the presented MdVNE approach is evaluated in four different scenarios and compared to a purely ILP-based technique. The following research questions are investigated in our experiments:

RQ 1: To what extent is the search space reduced by combining MT with ILP as compared to a purely ILP-based solution?

RQ 2: How does the reduction of the search space using MT affect the runtime of the ILP solver as compared to a purely ILP-based technique?

RQ 3: How does the reduction of the search space affect the embedding quality?

4.1 Setup

The evaluation setup consists of a two-tier SN and one or more VNs with a single-tier structure. The SN consists of 2 core switches with racks, each consisting of a switch and 10 servers (cpu = 64, memory = 128 and storage = 1000). The bandwidth between the servers and the switch in their rack is 1000, whereas the bandwidth between a rack and a core switch is 10000. The VNs consist of a central switch and range210 servers with varying resources (cpu = [2;12], memory = [2;24] and storage = [50;300]), which are interconnected (bandwidth = 100). For embedding, a pre-built queue with 100 randomly created VNs is used in the following four scenarios: Scenario S1 and S2 have 4 racks (Scenario S3 and S4 have 20 racks), and in each iteration either 1 VN (scenario S1 and S3) or 5 VNs simultaneously (scenario S2 and S4) are embedded. To calculate the costs for each mapping, the following values are defined as costs: Mapping a virtual switch to a substrate switch (server) costs 1 (2), and mapping a virtual server to a substrate server costs 1. The costs for mapping a virtual link to a substrate path are calculated depending on the path length (the number of links inside the path): a path length of 1 costs 2, while longer paths lead to a cost of $4^{\text{path length}}$ (e.g., a path length 2 leads to a cost of $4^2 = 16$). This strategy shall reflect the influence of resource consumption or latency on the quality of the embeddings. Finally, mapping a virtual link directly to a substrate node (server or switch) costs 1 (which is only possible if the virtual nodes connected by the virtual link are mapped to the same substrate node).

The following three configurations will be examined in more detail: *ILP*, *MdVNE A* and *MdVNE B*. *ILP* is purely based on the ILP problem formulation from Sect. 2. *MdVNE A* and *MdVNE B* demonstrate the possibilities of MdVNE to reduce the search space to different degrees. *MdVNE A* consists exclusively of rules of type (a) from Fig. 5, which check the type of the elements and ignore attribute constraints. *MdVNE B* also take into account the attributes and attribute constraints as depicted in the rules in Fig. 5(b).

The metrics defined by Fischer et al. [2] are used to measure quality. The first metric refers to energy consumption and measures the number of active substrate nodes (server and switches) in relation to all substrate nodes. An active element is defined as an element to which a virtual element is mapped. The second metric describes the average path length of a virtual link in hops, since each link has the same unit length. All simulations were run on a machine with Intel Xeon E5-2630 v3 with 2.40 GHz on Windows Server 2016 with Java SE Development Kit 8 and Gurobi 7.52 [11] as the ILP tool. The median of five repetitions is plotted in our runtime measurements.

4.2 Results

In the following, the results of the evaluation are presented on the basis of the research questions and discussed in detail.

RQ 1: Search Space. As a metric for the size of the search space we measured the mapping variables and constraints. Figure 7 present the number of mapping variables over the iterations for the scenarios S3 and S4 in the three configurations: *ILP*, *MdVNE A*, and *MdVNE B*. Both diagrams show that the *ILP* configuration requires about 7 times more variables than *MdVNE A* and *B* (e.g., Fig. 7, scenario S4: $\varnothing ILP = 290000$ and $\varnothing MdVNE\,B = 41000$). The reduced number of variables for *MdVNE B* in comparison to *MdVNE A* is due to the additional attribute constraints of *MdVNE B*, which limit the mapping possibilities. A similar behavior can also be observed in scenario S1 and S2. The number of constraints behave in the same way, whereby only 3 to 4 times more constraints are created for the *ILP* configuration compared to *MdVNE A* and *B*. The reduction of the numbers of constraints does not take place on the same scale as the reduction of the variables, since no duplicate constraints are filtered out in the translation $\mathcal{T}$.

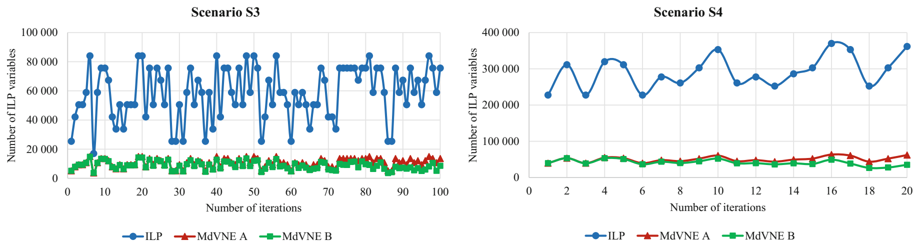
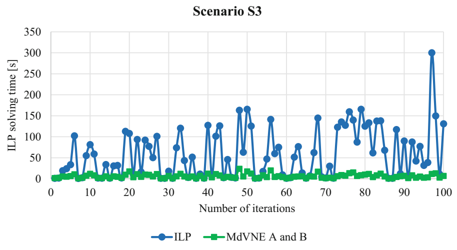


Fig. 7. ILP variables for scenario S3 and S4.

RQ 2: Runtime. Next, we assess the time required by Gurobi to solve the optimization problem as well as the entire runtime including preparation, MT, ILP solving, and post-processing. The two diagrams in Fig. 8 show the Gurobi runtime of scenario S3 and S4 (as scenario S1 and S2 behave similarly). Note that the Gurobi timeout is set to 2 h, which may not be sufficient to find an optimal result. It is shown that compared to *MdVNE A* and *B*, the *ILP* configuration takes on average about 9 times longer in scenario S3 than *MdVNE A* and *B* ($\varnothing ILP = 58$ and $\varnothing MdVNE\ B = 6$), and in scenario S4 the runtime even deviate by 2 orders of magnitude. Afterwards, the second measuring point in Fig. 8 (scenario S4) for the *ILP* configuration was evaluated again without any timeout for Gurobi resulting in approx. 8 h for the ILP solving. A similar behavior can also be observed in the scenarios S1 and S2, where the *ILP* configuration requires between 4 and 20 times longer to solve the ILP problem than *MdVNE A* and *B*. Figure 9 shows the complete runtime over the iterations for scenario S3, with the result that the *ILP* configuration takes approx. 4 times longer than *MdVNE A* and *B* ($\varnothing ILP = 62$ and $\varnothing MdVNE\,B = 17$). In scenario S4, this value is in the range of one order of magnitude ($\varnothing ILP = 6300$ and $\varnothing MdVNE\,B = 111$), whereby the Gurobi timeout must be taken into account. In order to illustrate the ratios of preparation, MT, and post-processing, Fig. 10 shows the complete runtime and the ILP solving time for scenario S4 and *MdVNE B* over the iterations. On average, the complete runtime for *MdVNE B* is 111 s and 67 s for ILP solving, so that in this case the ratio of MT and its preparation and post-processing takes about 40% of the complete time, which is in a comparable range as described in e.g. [12].



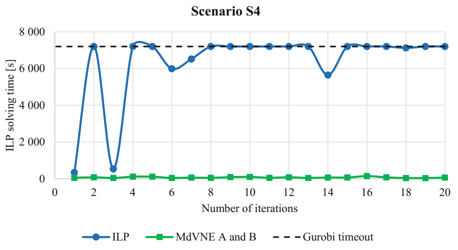
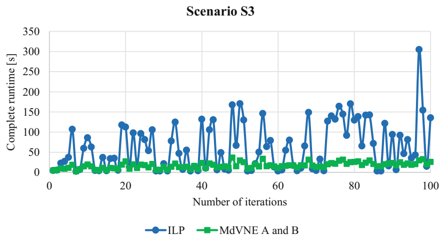


Fig. 8. ILP solving time for scenario S3 and S4.



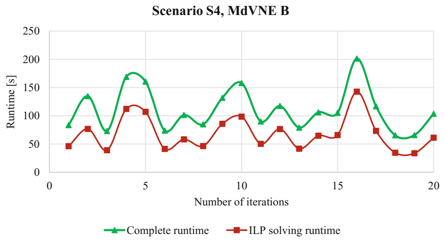


Fig. 9. Complete runtime for scenario S3.

Fig. 10. Complete and ILP solving runtime for *MdVNE B* in scenario S4.

RQ 3: Quality. To answer this question, the three metrics (i) accepted VNs, (ii) average path length and (iii) active substrate nodes were evaluated. The metrics accepted VNs and the average path length did not differ between the three configurations as expected, except in scenario S4, where the *ILP* ran into a timeout. Due to this, Gurobi was no longer able to find optimal solutions and the number of successfully mapped VNs decreased significantly. The last metric, counting the number of active substrate nodes, showed differences between the three configurations, as different optimal solutions are possible.

Summary. The evaluation showed that the search space of the embedding problem encoded as an ILP problem can be significantly reduced by the use of MTs, which also has a direct effect on the runtime for solving the optimization problem. Thus, *MdVNE A* and *B* required up to 7 times fewer variables and 4 times fewer constraints than the *ILP* configuration, resulting in a reduction of the ILP solution time by up to 2 orders of magnitude. When comparing the total runtime for *ILP* and *MdVNE A* and *B*, the *ILP* configuration required approximately one order of magnitude more time to solve the problem than *MdVNE A* and *B* including a ratio of about 40% for the MT in scenario S4. It also showed that the quality of the achieved solutions is comparable to the optimal solutions and that MT can, therefore, be specified in such a way that the search space can be significantly reduced without reducing the subset of optimal solutions.

Threats to Validity. Other technologies beside ILP are also possible to describe and solve the problem, e.g., SAT/SMT solver technologies. However, the ILP formulation used here is established [9] and is improved by an ongoing collaboration with experts. Required runtime for ILP solving, furthermore, highly depends on the choice of the ILP solver. Gurobi, nevertheless, represents a prominent and state-of-the-art solver, while we also plan further experiments with CPLEX.

5 Related Work

The virtualization of data center networks and the VNE problem has been extensively researched and an overview of these areas can be found in [2,13]. As a result, many algorithms have been developed for VNE to reduce the search space of this $\mathcal{NP}$-hard problem [1]. Guo et al. [4] proposed SecondNet, a heuristics-based approach to embed a subset of virtualized data centers in a tree-based data center. To reduce the search space, the authors only consider the bandwidth and number of virtual machines per physical server. Zeng et al. [5] additionally consider the data traffic between the virtual machines and minimize the resulting communication costs. The authors present an ILP-based approach for small data centers and a heuristics-based approach for larger data centers. Compared to the algorithms mentioned earlier, different topologies, resource constraints, requirements, and optimization goals can easily be taken into account by modifying MdVNEs metamodel and MT rules as well as the translation of generated models into sets of ILP formulas if necessary. Depending on the scenario, developers can reduce the search space and seamlessly adapt embedding decisions to changed boundary conditions while all constraints are fulfilled by construction, e.g., Fig. 5(b) $C_1^V \leq C_1^S$.

Model-driven software development is a promising method developing applications independently of a concrete plattform. The partly automatic verification of the specification and the automatic code generation also play an important role in a large number of different applications. For example, brake-by-wire in the automotive industry requires to allocate software components on networked electronic control units. Pohlmann et al. [14] describe a model-driven allocation approach specifying the problem in an OCL-based language, transformed into an ILP model and solved afterwards. In the area of Software-defined Networking, Lopes et al. [15] describe how to create application, controller, and network independent code for Software-defined Networking applications by modeling the physical network and its functions. Kluge et al. [16] present methods for the development of topology control algorithms by graph transformations taking into account global and local consistency constraints (e.g., preservation of connectivity). The aforementioned approaches indicate that model-driven development is a promising method for specifying algorithms in various network domains. Still, the focus of these models and approaches is not the simultaneous support for network resources and limitations or specifying VNE algorithms for data center environments.

Another approach for combining model transformation and optimization techniques is presented by Fleck et al. [17]. The objective is to calculate an "optimal" sequence of rule applications using search-based algorithms. Evaluating fitness values after each (arbitrarily performed) rule application, the approach reduces the search space on the fly but might fail in finding a global optimum. Another approach of optimization techniques in model-driven development is learning model transformations by examples [18], whereby the applicability to large models is the most limiting aspect.

6 Conclusion and Future Work

In this paper, we have introduced and expanded the MdVNE approach, which relies on a combination of MT and ILP techniques. This supports various resources, demands, constraints, and optimization goals while ensuring that all constraints are met by construction. Since the reduction of the search space is an important aspect in solving the VNE problem in data center, MdVNE enables the developer to reduce the search space by MT techniques according to purpose and scenario. This makes it possible to develop a series of embedding algorithms for scenarios on an abstract level, generate prototypical implementations automatically, and evaluate them rapidly within a reasonable time frame.

An evaluation of two MdVNE configurations and a pure ILP configuration for a data center of 40 and 200 servers was studied to evaluate the potential of reducing the search space, the runtime for problem solving and the quality of the embeddings. Our approach was able to reduce the search space by a factor of 7 compared to the pure ILP technique and thus the solver runtime can be reduced by up to 2 orders of magnitude. The complete runtime including the MT could be reduced by one order of magnitude in comparison to a purely ILP configuration. In addition, there were no significant differences in quality of the generated embeddings, so that MT was still able to find an optimal solution even after reducing the potential search space, compared to the ILP approach.

In future research, we plan to reduce the runtime in complex data center environments by using incremental pattern matching techniques. In addition, dynamic system changes should be taken into account, which means that migration and failure protecting strategies have to be integrated. This raises the question of the transition between different algorithms which, depending on the current situation, guarantee an optimal result and are adaptable to the respective environment. For further use of this approach, a simulation framework is currently being developed to test different configurations and new algorithms.

Acknowledgement. This work has been funded by the German Federal Ministry of Education and Research within the Software Campus project GraTraM at TU Darmstadt, funding code 01IS12054, and by the German Research Foundation (DFG) as part of project A1 within CRC 1053–MAKI.

References

1. Amaldi, E., Coniglio, S., Koster, A.M.C.A., Tieves, M.: On the computational complexity of the virtual network embedding problem. Electron. Notes Discrete Math. **52**, 213–220 (2016)
2. Fischer, A., Botero, J.F., Beck, M.T., de Meer, H., Hesselbach, X.: Virtual network embedding: a survey. Commun. Surv. Tutorials **15**(4), 1888–1906 (2013)
3. Ballani, H., Costa, P., Karagiannis, T., Rowstron, A.I.T.: Towards predictable datacenter networks. In: Conference on Applications, pp. 242–253 (2011)
4. Guo, C., Lu, G., Wang, H.J., Yang, S., Kong, C., Sun, P., Wu, W., Zhang, Y.: SecondNet: a data center network virtualization architecture with bandwidth guarantees. In: Proceedings of the 6th International Conference, pp. 15:1–15:12 (2010)
5. Zeng, D., Guo, S., Huang, H., Yu, S., Leung, V.C.: Optimal VM placement in data centers with architectural and resource constraints. Int. J. Auton. Adapt. Commun. Syst. **8**(4), 392–406 (2015)
6. Yang, Z., Guo, Y.: An exact virtual network embedding algorithm based on integer linear programming for virtual network request with location constraint. China Commun. **13**(8), 177–183 (2016)
7. Tomaszek, S., Leblebici, E., Wang, L., Schürr, A.: Model-driven development of virtual network embedding algorithms with model transformation and linear optimization techniques. In: Modellierung 2018, pp. 39–54 (2018)
8. Schrijver, A.: Theory of Linear and Integer Programming. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, New York (1999)
9. Sahhaf, S., Tavernier, W., Rost, M., Schmid, S., Colle, D., Pickavet, M., Demeester, P.: Network service chaining with optimized network function embedding supporting service decompositions. Comput. Netw. **93**, 492–505 (2015)
10. Schürr, A.: Specification of graph translators with triple graph grammars. In: Graph-Theoretic Concepts in Computer Science, pp. 151–163 (1994)
11. Gurobi Optimization, Inc.: Gurobi Optimizer Reference Manual 2015 (2016)
12. Leblebici, E., Anjorin, A., Schürr, A.: Inter-model consistency checking using triple graph grammars and linear optimization techniques. In: Fundamental Approaches to Software Engineering, pp. 191–207 (2017)
13. Bari, M.F., Boutaba, R., Esteves, R.P., Granville, L.Z., Podlesny, M., Rabbani, M.G., Zhang, Q., Zhani, M.F.: Data center network virtualization: a survey. Commun. Surv. Tutorials **15**(2), 909–928 (2013)
14. Pohlmann, U., Hüwe, M.: Model-driven allocation engineering (T). In: International Conference on Automated Software Engineering, pp. 374–384 (2015)
15. Lopes, F.A., Lima, L., Santos, M., Fidalgo, R., Fernandes, S.: High-level modeling and application validation for SDN. In: Network Operations and Management Symposium, pp. 197–205 (2016)
16. Kluge, R., Stein, M., Varró, G., Schürr, A., Hollick, M., Mühlhäuser, M.: A systematic approach to constructing incremental topology control algorithms using graph transformation. J. Vis. Lang. Comput. **38**, 47–83 (2017)

17. Fleck, M., Troya, J., Wimmer, M.: Marrying search-based optimization and model transformation technology. In: Proceedings of NasBASE (2015)
18. Kessentini, M., Sahraoui, H., Boukadoum, M.: Model transformation as an optimization problem. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 159–173. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87875-9_12

Schema Transformations and Query Rewriting in Ontological Databases with a Faceted Interface

Tadeusz Pankowski(✉)

Institute of Control, Robotics and Information Engineering,
Poznań University of Technology, Poznań, Poland
tadeusz.pankowski@put.poznan.pl

Abstract. In this paper, we discuss some problems identified in designing and implementing a class of ontological database systems. The goal of these systems is to provide an extended knowledge system that combines flexibility of ontologies with efficiency of relational databases. The terminological part of the ontology forms the ontological (conceptual) schema of the database, and the extensional part is managed by a relational database server. Queries are formulated in an interactive way using a faceted search over the ontological schema. In such scenario, a number of transformations must be performed: (a) a mapping from an ontological schema into relational scheme that concerns both the structure and rules constituting the ontology; (b) transformation of faceted queries, defined in a graphical form, into first-order queries and to SQL queries. The considerations are based on verified solutions implemented in DAFO (*Data Access based in Faceted queries over Ontologies*) system.

1 Introduction

In recent two decades, we witness the increasing importance of research on combining ontologies and databases. Ontology-based data access (OBDA) systems have been proposed as a way of integrating and querying information from heterogeneous sources [6,21], in ontology-enhanced databases an ontology is added to enrich databases with intensional knowledge encoded by ontology rules [2,13,19], ontological databases are designed for ontologies, where query answering problems are more important than classical reasoning tasks [10]. In all these cases, the goal is to achieve a synergy in the result of combining flexibility of ontologies with efficiency of relational databases. The challenging issue is providing the system with a querying mechanism over ontologies, since the schema of the database has then a form of an ontology. A standard query language for ontologies is SPARQL [22] but it is not well-suited for end users. Thus, we observe development of graphic-oriented query languages [26]. Among them, a significant place takes the faceted search [2,24,25], and in this paper we will exploit this paradigm as a query formulation mechanism in ontological database systems.

A. Rensink and J. Sánchez Cuadrado (Eds.): ICMT 2018, LNCS 10888, pp. 76–91, 2018.
https://doi.org/10.1007/978-3-319-93317-7_3

Contribution. In this paper, we discuss and propose solutions to schema transformations and query rewriting problems identified in designing and implementing in ontological database systems with faceted search. Main novelties discussed in this paper, are:

1. Defining a mapping between a terminological part of an ontology (treated as a conceptual schema) and a schema of relational database. This mapping, called meta-schema mapping, since is defined on the meta-schema level and concerns schemas, specifies a mapping for both structural elements of the ontology as well as ontology rules. This is crucial in ontological databases since the mapping is used in the process of translating ontology-oriented queries into SQL queries executed by relational database engine.
2. Designing a transformation and rewriting procedures from ontology-oriented faceted queries into first-order queries and finally to SQL queries. A faceted query is created using a hierarchical graphical interface and is visualized as a tree. We propose a method for transforming such a tree in a first-order query, which is a tree-shaped monadic positive existential query [1,2], that is next translated into a SQL query.

Related work. Similarities and differences between ontologies and relational databases were investigated in both research papers [13,18] and text books, notably [1]. A synergic combination of databases and ontologies can be observed in data integration systems [6,21], ontology-enhanced databases [2,13,19] and ontological databases [4,5,10]. Essential and inspiring for this paper was the concept of the extended knowledge bases proposed in [13]. The theory of schema mapping was initiated and developed in [8,9]. Faceted search is a prominent search and data exploration paradigm in ontology-based and e-commerce applications and a number of RDF-based faceted search systems have been developed in recent years [2,12,20,25,27]. The theoretical foundations of faceted search were studied [2]. In our previous papers [15,16] we presented the preliminary results of our investigations and implementation of the *Data Access based on Faceted queries over Ontologies* (DAFO) system and reported experimental results, which prove high efficiency of proposed solutions.

2 Preliminaries: Ontological Databases

An application domain can be specified as an ontology, since an ontology is *"an explicit specification of a conceptualization of an application domain"* [11]. On the other hand, from a more technical point of view, an application domain is represented using the relational database technology, where the conceptual model of the application domain is informally presented in a form of ER or UML diagram (see our running example in Fig. 1). The idea of *ontological databases* is to combine these two approaches. Ontologies provide a rich mechanism for intensional knowledge specification (the conceptual model), and relational database technology provides methods for efficient implementation of the extensional knowledge (a relational database) and a query answering environment.

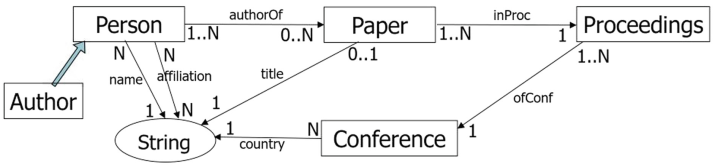


Fig. 1. ER diagram as a conceptual model of a bibliographic application domain.

2.1 Ontologies and Ontological Schemas

An ontology is defined as a triple $\mathcal{O} = (\Sigma, \mathcal{R}, \mathcal{A})$, where: (a) Σ is a *signature* consisting of a set of unary predicates (UP) (classes) and binary predicates (BP) (properties), (b) $\mathcal{R}$ is a set of *rules* (intensional knowledge), and (c) $\mathcal{A}$ is a set of *facts* or assertions. All expressions in $\mathcal{R}$ and in $\mathcal{A}$ are built from symbols occurring in Σ and constants in a Const set. We assume that there is a subset $\mathsf{LabNull}$ of *labeled nulls* in Const. For labeled nulls, the *unique name assumption* (UNA) is not made, i.e., different symbols can denote the same individual. For "regular" constants UNA is satisfied [1,8,13]. A pair $\mathcal{T} = (\Sigma, \mathcal{R})$ is referred to as a *terminological part* of the ontology, and $\mathcal{A}$ as an extensional part of it [3].

In ontological databases, the intensional part plays a role of an *ontological (conceptual) schema* presented to users, while the extensional part is represented in a relational database $\mathcal{R} = (Sch, I)$, where Sch is a relational schema and I is an instance of the schema [10]. In this paper, we impose some restrictions on the terminological part to achieve ability to transform ontological schema into the relational one.

Definition 1 (Ontological schema). *A pair $\mathcal{T} = (\Sigma, \mathcal{R})$ is an ontological schema (a schema of an ontological database), if:*

1. *Σ is partitioned into two disjoint sets of intensional and extensional predicates, i.e., $\Sigma = \mathsf{UP} \cup \mathsf{BP}$, where: $\mathsf{UP} = \mathsf{UP}_I \cup \mathsf{UP}_E$, and $\mathsf{BP} = \mathsf{BP}_I \cup \mathsf{BP}_E$, and the distinguished extensional unary predicate String is in UP_E.*
2. *The set of rules is partitioned into two disjoint sets, $\mathcal{R} = \mathcal{R}_c \cup \mathcal{R}_w$, Table 1, where:*
   - *$\mathcal{R}_c$ – a set of integrity constraint rules of the form* (IC-1) – (IC-8)*.*
   - *$\mathcal{R}_w$ – a set of rewriting rules of the form* (RW-1) – (RW-5)*. All predicates on the right-hand sides of rewriting rules are intensional.*

Definition 1 determines a class of ontologies which can be transformed into relational databases without losing knowledge, and which can be queried using a faceted search paradigm.

Example 1. The conceptual model represented by an ER diagram in Fig. 1, can be specified by an ontological schema $BibOn = (\Sigma, \mathcal{R}_c \cup \mathcal{R}_w)$, where:

Table 1. Categories of rules in DAFO ontologies.

Id	General form of a rule	Name
IC-1	$P(x,y) \rightarrow C(x)$	Domain rule
IC-2	$P(x,y) \rightarrow C(y)$	Range rule
IC-3	$C(x) \rightarrow C_1(x)$	Subtype rule (follows from RW-3 or RW-4)
IC-4	$P(x,y) \rightarrow P_1(x,y)$	Subproperty rule (follows from RW-5)
IC-5	$C(x) \rightarrow \exists y\ P(x,y)$	Totality on domain
IC-6	$C(y) \rightarrow \exists x\ P(x,y)$	Totality on range
IC-7	$P(x,y_1) \wedge P(x,y_2) \rightarrow y_1 = y_2$	Functionality rule (a function on domain)
IC-8	$P(x_1,y) \wedge P(x_2,y) \rightarrow x_1 = x_2$	Key rule (a function on range)
RW-1	$\exists z\ P_1(x,z) \wedge C(z) \wedge P_2(z,y) \rightarrow P(x,y)$	Chain (composition)
RW-2	$P_1(y,x) \rightarrow P(x,y)$	Inversion
RW-3	$C_1(x) \wedge \exists y P(x,y) \wedge y = a \rightarrow C(x)$	Type specialization determined by a value
RW-4	$C_1(x) \wedge \exists y P(x,y) \wedge C_2(y) \rightarrow C(x)$	Type specialization determined by a subtype
RW-5	$C_1(x) \wedge P_1(x,y) \wedge C_2(y) \rightarrow P(x,y)$	Property specialization determined by a domain and/or a range subtype

1. Extensional predicates:
   $\mathsf{UP}_E = \{String, Person, Paper, Proceedings, Conference\}$
   $\mathsf{BP}_E = \{name, affiliation, authorOf, inProc, ofConf, country, acronym \ldots\}$.
2. Intensional predicates (not all are depicted in Fig. 1):
   $\mathsf{UP}_I = \{Author, ACMConf, ACMPaper, \ldots\}$,
   $\mathsf{BP}_I = \{writtenBy, presentedAt, authorConf, \ldots\}$.
3. Integrity constraint rules (only concerning *name*):

$$\begin{aligned}\mathcal{R}_c = \{&name(x,y) \rightarrow Person(x), Person(x) \rightarrow \exists y\ name(x,y),\\ &name(x,y_1) \wedge name(x,y_2) \rightarrow y_1 = y_2,\\ &name(x_1,y) \wedge name(x_2,y) \rightarrow x_1 = x_2, \ldots\}\end{aligned}$$

4. Rewriting rules specifying intensional predicates:

$$\begin{aligned}\mathcal{R}_w = \{&Paper(x) \wedge \exists y\ authorOf(x,y) \wedge Paper(y) \rightarrow Author(x),\\ &Conference(x) \wedge \exists y\ acronym(x,y) \wedge y =' ACM' \rightarrow ACMConf(x),\\ &Paper(x) \wedge \exists y\ presentedAt(x,y) \wedge ACMConf(y) \rightarrow ACMPaper(x)\\ &authorOf(y,x) \rightarrow writtenBy(x,y),\\ &\exists z\ inProc(x,z) \wedge Proceedings(z) \wedge ofConf(z,y) \rightarrow presentedAt(x,y),\\ &\exists z\ authorOf(x,z) \wedge Paper(z) \wedge presentedAt(z,y) \rightarrow authorConf(x,y)\}\end{aligned}$$

2.2 Relational Schema

A *relational schema* specifies relation names, their types (set of attributes), primary keys for some relations, and such integrity constraints as: primary key constraints, unique constraints, not-null constraints, referential constraints (foreign keys), and inclusion constraints.

Definition 2 (Relational schema). *A relational schema is a tuple* $Sch = (\mathbf{R}, \mathsf{Att}, att, pkey, \mathsf{Constr})$, *where:*

- $\mathbf{R} = \{R_1, \ldots, R_n\}$ *is a finite set of relation (or table) names;*
- Att *is a finite set of attributes;*
- *att assigns a finite set* $att(R) \subseteq \mathsf{Att}$ *of attributes to each* $R \in \mathbf{R}$*;*
- *pkey is a function assigning primary keys to some relation names,* $pkey(R) = Id \in att(R)$ *(i.e., we assume that all primary keys have the same name, Id);*
- Constr *is a set of constraints of the form:* $R[Id]$ `PRIMARY KEY` *(primary key constraint);* $R[A]$ `UNIQUE` *(unique constraint);* $R[A]$ `NOT NULL` *(not-null constraint);* $R[A] \rightarrow R'[Id]$ *(referential constraint);* $R'[Id] \subseteq R[A]$ *(inclusion constraint), where* $A \in att(R)$, $Id = pkey(R')$, $R, R' \in \mathbf{R}$.

Example 2. A sample relational schema of a bibliographic database is depicted in Fig. 2. There are six relation names, and, for example:

$$\begin{aligned}
&att(Person) &&= \{Id, Name\},\\
&att(Affiliation) &&= \{PersonId, Affiliation\},\\
&pkey(Person) &&= Id,\\
&\mathsf{Constr} &&= \{Person[Id]\ \texttt{PRIMARY KEY}, Person[Name]\ \texttt{UNIQUE},\\
& && \quad Person[Name]\ \texttt{NOT NULL}, AuthorPaper[AuthorId]\ \texttt{NOT NULL},\\
& && \quad AuthorPaper[AuthorId] \rightarrow Person[Id],\\
& && \quad Paper[Id] \subseteq AuthorPaper[PaperId], \ldots\}.
\end{aligned}$$

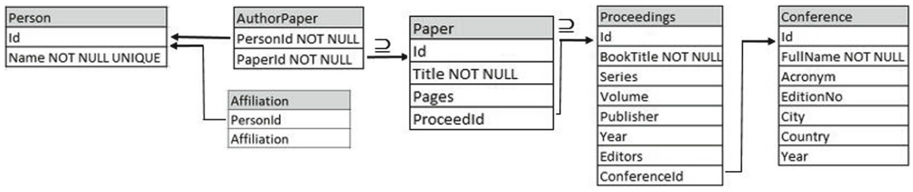


Fig. 2. Diagram of a bibliographic relational schema BibSch. Arrows denote referential constraints, and $\subseteq$ – inclusion constraints.

3 Transforming of an Ontology to Ontological Database

Given an ontology $\mathcal{O} = (\Sigma, \mathcal{R}_c \cup \mathcal{R}_w, \mathcal{A})$, we have to transform it to a relational database $RDB = (Sch, I)$. So, two mappings must be defined:

- on a meta-schema level: a mapping of ontological to relational schema,
- on a schema level: a mapping of a set of facts, extended with materialized extensional rules, to relational database instance.

In result, and ontological database $ODB = (\Sigma, \mathcal{R}_c \cup \mathcal{R}_w, RDB)$ is obtained.

3.1 Mapping of Ontological Schema to Relational Schema

Now, we show how elements of an ontological schema are mapped to elements of a relational schema. Let $\mathcal{T} = (\Sigma, \mathcal{R}_c \cup \mathcal{R}_w)$ be an ontological schema. The set $\mathsf{BP}_E \subseteq \Sigma$ of extensional binary predicates is divided into four pairwise disjoint sets:

$$\begin{array}{ll} FDP & -\text{a set of functional data properties,} \\ MDP & -\text{a set of multivalued data properties,} \\ FOP & -\text{a set of functional object properties,} \\ MOP & -\text{a set of multivalued object properties.} \end{array}$$

Data properties are binary predicates which have *String* as their range, otherwise they are *object properties*. A *functional property* is a binary predicate for which the functionality rule is defined, otherwise it is a *multivalued* property. Further on, by $dom(P)$ and $rng(P)$ we denote, respectively, the domain and the range of P, by $domTot(P)$ and $rngTot(P)$ we denote that P is total on its domain and range, respectively, and by $key(P)$, that the key rule is defined for P.

Definition 3. *Given an ontological schema $\mathcal{T} = (\Sigma, \mathcal{R}_c \cup \mathcal{R}_w)$ and a relational schema $Sch = (\mathbf{R}, \mathsf{Att}, att, pkey, \mathsf{Constr})$, a (meta-schema) mapping from $\mathcal{T}$ to Sch is defined by means of the following meta-schema mapping function*

$$M = (rel, domAtt, rngAtt, constr),$$

where

$$\begin{array}{ll} rel & : \mathsf{UP}_E \cup \mathsf{BP}_E \to \mathbf{R} \\ domAtt & : \mathsf{BP}_E \to \mathsf{Att} \\ rngAtt & : \mathsf{BP}_E \to \mathsf{Att} \\ constr & : \mathsf{UP}_E \cup \mathsf{BP}_E \to 2^{\mathsf{Constr}}. \end{array}$$

We assume the following denotations:

– $rel(C)$, and $rel(P)$ will be denoted by R_C and R_P, respectively;
– $domAtt(P)$ will be denoted by A_P^d;
– $rngAtt(P)$ will be denoted by A_P^r.

The above meta-schema mapping function M is defined as follows:

1. For every $C, C' \in \mathsf{UP}_E$, we assume $R_C \neq R_{C'}$ for $C \neq C'$, and

$$R_C[Id]\ \texttt{PRIMARY KEY} \in constr(C).$$

   To any extensional unary predicate C, a relation name R_C of type $att(R_C)$ is assigned. There is an attribute $Id \in att(R_C)$, and Id is the primary key of R_C. Different relation names are assigned to different unary predicates. For example: $R_{Person} = Person$, $Id \in att(Person)$, $pkey(Person) = Id$, $Person[Id]$ `PRIMARY KEY` $\in \mathsf{Constr}$.

2. For every $P \in FDP$, if $dom(P) = C$, then
$$R_P = R_C,\ A_P^d = Id,\ A_P^r \in att(R_C),\ A_P^r \neq Id.$$
If P is a functional data property with domain C, then: an attribute of R_C is assigned to P as its range attribute A_P^r, and the primary key of R_C is assigned to P as its domain attribute A_P^d. The mapping does not effect the set of constraints in the relational schema. For example, if $name \in FDP$, $dom(name) = Person$, then $R_{name} = Person$, $A_{name}^d = Id$, $A_{name}^r = Name \in att(Person)$.
3. For every $P \in MDP$, if $dom(P) = C$, then:
$$R_P \neq R_C,\ att(R_P) = \{A_P^d, A_P^r\},\ R_P[A_P^d] \to R_C[Id] \in constr(P).$$
If P is a multivalued data property with domain C, then: a relation name R_P assigned to P is different from R_C; relation R_P has two attributes, $att(R_P) = \{A_P^d, A_P^r\}$, assigned as the domain and the range attribute of P. The domain attribute is the foreign key in R_P referring to the primary key of R_C, i.e., the referential constraint $R_P[A_P^d] \to R_C[Id]$ is assigned to P. For example, if *affiliation* is a multivalued data property with domain $Person$, then $R_{affiliation} = Affiliation$, $att(Affiliation) = \{PersonId, Affiliation\}$, $A_{affiliation}^d = PersonId$, $A_{affiliation}^r = Affiliation$, and $Affiliation[PersonId] \to Person[Id] \in \mathsf{Constr}$.
4. For every $P \in FOP$, if $dom(P) = C$, and $rng(P) = D$, then
$$R_P = R_C,\ A_P^d = Id,\ A_P^r \in att(R_C),\ R_C[A_P^r] \to R_D[Id] \in constr(P).$$
If P is a functional object property with domain C and range D, then the relation assigned to P is that assigned to C. The domain attribute of P is the primary key of R_C, and the range attribute of P is an attribute $A_P^r \in att(R_C)$. A_P^r is also a foreign key referring to the primary key of R_D. For example, if $ofConf$ is a functional object property with domain $Proceedings$ and range $Conference$, then: $R_{ofConf} = Proceedings$, $A_{ofConf}^d = Id = pkey(Proceedings)$, $A_{ofConf}^r = ConferenceId \in att(Proceedings)$, and the referential constraint $Proceedings[ConferenceId] \to Conference[Id]$ is in Constr.
5. For every $P \in MOP$, if $dom(P) = C$, and $rng(P) = D$, then:
$$R_P \neq R_C,\ R_P \neq R_D,\ att(R_P) = \{A_P^d, A_P^r\},$$
$$\{R_P[A_P^d] \to R_C[Id],\ R_P[A_P^r] \to R_D[Id]\} \subseteq constr(P).$$
If P is a multivalued object property with domain C and range D, then the relation assigned to P is a binary relation R_P, different from R_C and R_D. $att(R_P) = \{A_P^d, A_P^r\}$, where A_P^d is a foreigns key referring to the primary key of R_C, and A_P^r is a foreign key referring to the primary key of R_D. For example, if $authorOf$ is a multivalued object property with domain $Person$ and range $Paper$, then: $R_{authorOf} = AuthorPaper$, $att(AuthorPaper) = \{AuthorId, PaperId\}$, $A_{authorOf}^d = AuthorId$, $A_{authorOf}^r = PaperId$ and the referential constraints $AuthorPaper[AuthorId] \to Person[Id]$ and $AuthorPaper[PaperId] \to Paper[Id]$ are in Constr.

6. *Mapping of another integrity constraint rules.* Note that the following integrity constraints rules: domain, range, and functionality, were considered while defining the above mappings. The subtype and subproperty rules are not mapped explicitly, since they are in fact consequences of some rewriting rules. Now, we define mappings for totality and key rules.
   (a) If $dom(P) = C$, and $domTot(P)$ is a totality rule on domain of P, then

   $$\{R_P[A_P^r]\ \texttt{NOT NULL},\ R_C[Id] \subseteq R_P[A_P^d]\} \subseteq constr(P).$$

   In this case, the rule is mapped into two constraints in relational schema: (1) the not-null constraint on the range attribute of P, and (2) the inclusion dependency defining inclusion of the primary key of R_C in the domain attribute of P (trivially satisfied for functional properties).
   - $Person(x) \rightarrow \exists y\ name(x, y)$ saying that $name$ is total on $Person$ is mapped to $Person[Name]$ `NOT NULL`, and to $Person[Id] \subseteq Person[Id]$ (that is always true).
   - The rule $Person(x) \rightarrow \exists y\ affiliation(x, y)$ says that at least one affiliation is given for any person. Then *affiliation* is total on $Person$, and the rule is mapped into the set of two constraints:
     (1) the not-null constraint: $Affiliation[Affiliation]$ `NOT NULL`, and
     (2) the inclusion dependency: $Person[Id] \subseteq Affiliation[PersonId]$.

   (b) If $rng(P) = D$, and $rngTot(P)$ is a totality rule on range of P, then

   $$\{R_P[A_P^d]\ \texttt{NOT NULL},\ R_D[Id] \subseteq R_P[A_P^r]\} \subseteq constr(P).$$

   In this case, the rule is mapped into two constraints in relational schema: (1) the not-null constraint on the domain attribute of P (trivially satisfied for functional properties, since the domain attribute is then the primary key), and (2) the inclusion dependency defining inclusion of the primary key of R_D in the range attribute of P.
   - $Conference(y) \rightarrow \exists x\ ofConf(x, y)$ specifies that any conference has proceedings, i.e., $ofConf$ is total on its range $Conference$. The rule is mapped to $Proceedings[Id]$ `NOT NULL` (always true), and to inclusion dependency $Conference[Id] \subseteq Proceedings[ConferenceId]$.
   - The rule $Paper(y) \rightarrow \exists x\ authorOf(x, y)$ says that at least one author must be given for any paper. Then *authorOf* is total on *Paper*, and the rule is mapped into the set of two constraints:
     (1) the not-null constraint: $AuthorPaper[AuthorId]$ `NOT NULL`, and
     (2) the inclusion dependency: $Paper[Id] \subseteq AuthorPaper[PaperId]$.
7. If $key(P)$ is a key rule, then

   $$R_P[A_P^r]\ \texttt{UNIQUE} \in constr(P),$$

   i.e., the `UNIQUE` constraint is enforced for the range attribute assigned to P. For example, if $name(x_1, y) \wedge name(x_2, y) \rightarrow x_1 = x_2$ says that a person is uniquely determined by its name, then this rule is mapped to the constraint $Person[Name]$ `UNIQUE` in relational schema.

Meta-schema mappings for some predicates of the ontological schema in Example 1 into relational schema in Example 2, is given in Table 2.

Table 2. Meta-schema mapping of an ontological schema into relational schema.

Predicate (P)	*rel* (R_P)	*domAtt* (A^d_P)	*rngAtt* (A^r_P)	*constr*(P)
Person(UP)	Person			{Person[Id] PRIMARY KEY}
name(FDP)	Person	Id	Name	{Person[Name] NOT NULL, Person[Name] UNIQUE}
affiliation(MDP)	Affiliation	PersonId	Affiliation	{Affiliation[PersonId] → Person[Id]}
inProc (FOP)	Paper	Id	ProcId	{Paper[ProcId] → Proceedings[Id], Proceedings[Id] ⊆ Paper[ProcId]}
authorOf (MOP)	AuthorPaper	AuthorId	PaperId	{AuthorPaper[AuthorId] → Person[Id], AuthorPaper[PaperId] → Paper[Id], Paper[Id] ⊆ AuthorPaper[PaperId], AuthorPaper[AuthorId] NOT NULL, AuthorPaper[PaperId] NOT NULL}

3.2 Mapping of an Ontology to Relational Database Instance

Now, we show haw extensional part of an ontology $\mathcal{O} = (\Sigma, \mathcal{R}_c \cup \mathcal{R}_w, \mathcal{A})$ is mapped to an relational database instance. This extensional part consists of the set $\mathcal{A}$ and all consequences of it with respect to the set $\mathcal{R}_c$ of integrity constraint rules, denoted $\mathcal{A} \cup \mathcal{R}_c$. This extensional part, called the canonical model, is obtained by using the *chase procedure* [7,8], denoted $\mathcal{A} \cup \mathcal{R}_c = chase_{\mathcal{R}_c}(\mathcal{A})$.

Example 3. For the following sets from an ontology $\mathcal{O}$:

– a set of facts (N_i denotes a labeled null in $\mathsf{LabNull}$)
 $\mathcal{A} = \{Person(\mathsf{N}_1), Person(p_2), name(\mathsf{N}_1, john),$
 $name(\mathsf{N}_2, ann), name(p_1, john)\}$, $\mathsf{N}_1, \mathsf{N}_2 \in \mathsf{LabNull}$, and
– a set of integrity constraint rules
 $\mathcal{R}_c = \{name(x, y) \to Person(x),\ Person(x) \to \exists y\ name(x, y),$
 $name(x_1, y) \wedge name(x_2, y) \to x_1 = x_2\}$;

it is easy to show that the canonical model is

$$\mathcal{A} \cup \mathcal{R}_c = \{Person(p_1), Person(p_2), Person(\mathsf{N}_2), \\ name(p_1, john), name(p_2, \mathsf{N}_3), name(\mathsf{N}_2, ann)\}.$$

Definition 4. *Let $\mathcal{O} = (\mathcal{T}, \mathcal{A})$ be an ontology, where $\mathcal{T} = (\Sigma, \mathcal{R}_c \cup \mathcal{R}_w)$, and $M : \mathcal{T} \to Sch$ be a mapping from the ontological schema $\mathcal{T}$ to a relational schema Sch. A mapping, m, of the extensional component of $\mathcal{O}$ to an instance of Sch, is defined by the following set of rules executed on the canonical model $\mathcal{A} \cup \mathcal{R}_c$:*

– $C(x) \to \exists r\ R_C(r) \wedge r.Id = x$, *for each* $C \in \mathsf{UP}_E$*;*
– $P(x, y) \to \exists r\ R_P(r) \wedge r.A^d_P = x \wedge r.A^r_P = y$, *for each* $P \in \mathsf{BP}_E$*;*
– $R_P(r) \wedge isLN(r.A^r_P) \to r.A^r_P = \mathsf{NULL}$, *for each data property* P*.*

where R_C, R_P, A_P^d, A_P^r *are defined in the definition of* M *(Definition 3). The function* $isLN(x)$ *returns* TRUE *if* $x \in \mathsf{LabNull}$, *and* FALSE *otherwise.*

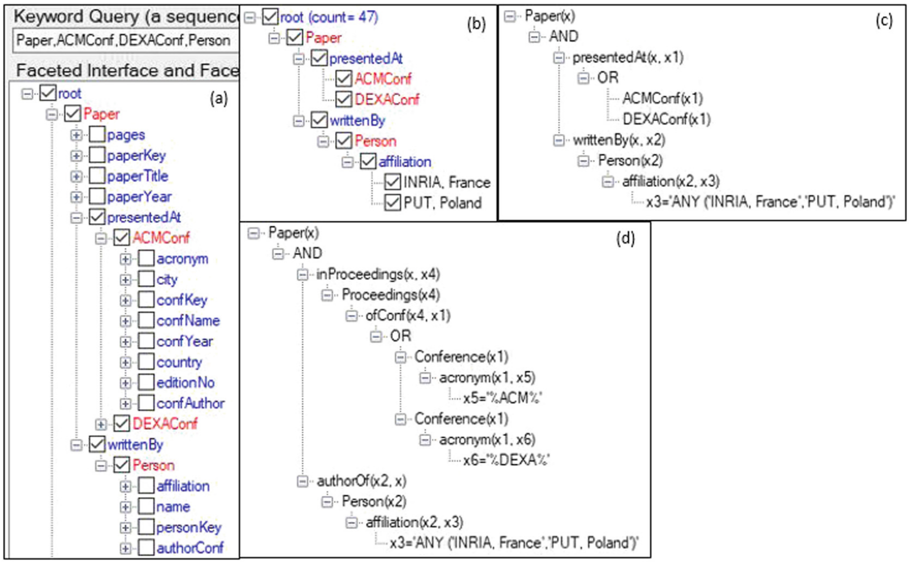
Note, that the right-hand sides of rules in m (Definition 4) are specified in *relational tuple calculus* (RTC) [1]. Labeled nulls, which are values of data properties, are replaced by NULL, while the other are left in the relational database instance and are interpreted as "regular" constants (satisfying UNA).

Example 4. For the canonical model in Example 3, and the mapping discussed in Sect. 3.1, we obtain $m(\mathcal{A} \cup \mathcal{R}_c) = I$, where
$Person^I = \{[Id : p_1, name : john], [Id : p_2, name : \mathtt{NULL}], [Id : \mathsf{N}_2, name : ann]\}$.

4 Faceted Queries and Their First-Order Form

4.1 Formulating Faceted Queries Using a Faceted Interface

The main assumption in the faceted search paradigm is that a user is provided with a faceted interface that is a hierarchical view over the underlying information source, in our case – over an ontology with the ontological schema $\mathcal{T} = (\Sigma, \mathcal{R}_c \cup \mathcal{R}_w)$. An ontology is, in general, a complex network not a hierarchy. To deal with this issue, in DAFO system a user starts with writing a keyword query that is a sequence of unary predicates from the ontology. In this way she/he specifies an interesting subset of the ontology and its hierarchical arrangement (consistent with the ordering of the keywords in the sequence).



```
Keyword Query (a sequenc
Paper,ACMConf,DEXAConf,Person
Faceted Interface and Face
☑ root                         (a)
  ☑ Paper
    ☐ pages
    ☐ paperKey
    ☐ paperTitle
    ☐ paperYear
    ☑ presentedAt
      ☑ ACMConf
        ☐ acronym
        ☐ city
        ☐ confKey
        ☐ confName
        ☐ confYear
        ☐ country
        ☐ editionNo
        ☐ confAuthor
      ☑ DEXAConf
    ☑ writtenBy
      ☑ Person
        ☐ affiliation
        ☐ name
        ☐ personKey
        ☐ authorConf
```

```
☑ root (count= 47)             (b)
  ☑ Paper
    ☑ presentedAt
      ☑ ACMConf
      ☑ DEXAConf
    ☑ writtenBy
      ☑ Person
        ☑ affiliation
          ☑ INRIA, France
          ☑ PUT, Poland
```

```
Paper(x)                       (c)
  AND
    presentedAt(x, x1)
      OR
        ACMConf(x1)
        DEXAConf(x1)
    writtenBy(x, x2)
      Person(x2)
        affiliation(x2, x3)
          x3='ANY ('INRIA, France','PUT, Poland')'
```

```
Paper(x)                       (d)
  AND
    inProceedings(x, x4)
      Proceedings(x4)
        ofConf(x4, x1)
          OR
            Conference(x1)
              acronym(x1, x5)
                x5='%ACM%'
            Conference(x1)
              acronym(x1, x6)
                x6='%DEXA%'
    authorOf(x2, x)
      Person(x2)
        affiliation(x2, x3)
          x3='ANY ('INRIA, France','PUT, Poland')'
```

Fig. 3. A faceted interface (a), faceted query tree (b), and first-order faceted query before rewriting (c), and after rewriting (d). (Color figure online)

For example, $(Paper, ACMConf, DEXAConf, Person)$ is the keyword query in Fig. 3 that initiates a process of the faceted search. It means that the user intends to create a query returning a set of objects of type *Paper* (the first component of the sequence), and the query will involve conditions on remaining components of the sequence. The system prepares an appropriate hierarchical view of the ontology and presents it in a form of a *faceted interface*, Fig. 3(a). Note that all unary predicates given in the keyword query, as well as binary predicates connecting them, are selected (checked). In DAFO, a faceted interface is implemented by a *tree view* object that is an instance of *TreeView class* [23].

Now, a user can interactively and iteratively operate on the faceted interface creating the expected final *faceted query*. In the running example, the faceted query is presented as the tree in Fig. 3(b) (*count* = indicates the expected number of elements in the answer). Note that:

– the blue color of a node indicates that the set of its children is a *disjunctive set* (components are connected by `OR`),
– the red color of a node indicates that the set of its children forms a *conjunctive set* (components are connected by `AND`),
– a disjunctive set of constants (or constant set names) is written as a list and prefixed by `ANY`, e.g., *ANY ('INRIA France', 'PUT Poland')* in Fig. 3(c),
– a conjunctive set of constants (or constant set names) is written as a list and prefixed by `ALL`, e.g., *ALL ('%database%', '%query%')* in Fig. 4(b), where *%database%* identifies a set of strings containing *'database'* as a substring.

While operating on the faceted interface, a user can perform the following operations: (a) selecting/unselecting nodes, (b) expanding/collapsing subtrees, (c) switching: disjunctive/conjunctive, (d) cloning (duplicating) subtrees, (e) inserting values, (f) removing unselected nodes.

In result of operating over a faceted interface, a final faceted query is obtained. In Fig. 3(b) the faceted query is presented as a tree. The meaning of the query is: *"Get papers presented at an ACM or a DEXA conference and written by a person affiliated to 'INRIA France' or 'PUT Poland'"*.

The faceted query in Fig. 4(a) means: "Get ACM papers, which concerns both databases and queries" (for simplicity, we assume that the subjects of a paper are included in its title).

4.2 Transformation of Faceted Queries into First-Order Form

To obtain an executable form of a faceted query, we have to make the following transformations:

1. Transformation of the tree form of faceted query into first-order form, called *first-order faceted query* (FOFQ).
2. Rewriting FOFQ into an extensional form, i.e., into a form in which no intensional predicate occurs.

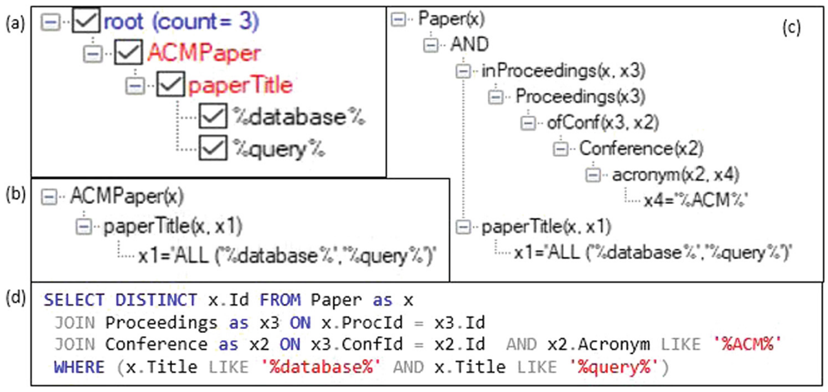

```
(a)
root (count= 3)
  ACMPaper
    paperTitle
      %database%
      %query%
```
```
(b)
ACMPaper(x)
  paperTitle(x, x1)
    x1='ALL ("%database%","%query%")'
```
```
(c)
Paper(x)
  AND
    inProceedings(x, x3)
      Proceedings(x3)
        ofConf(x3, x2)
          Conference(x2)
            acronym(x2, x4)
              x4='%ACM%'
    paperTitle(x, x1)
      x1='ALL ("%database%","%query%")'
```
```
(d)
SELECT DISTINCT x.Id FROM Paper as x
 JOIN Proceedings as x3 ON x.ProcId = x3.Id
 JOIN Conference as x2 ON x3.ConfId = x2.Id  AND x2.Acronym LIKE '%ACM%'
 WHERE (x.Title LIKE '%database%' AND x.Title LIKE '%query%')
```

Fig. 4. Faceted query tree (a), first-order faceted query before rewriting (b), after rewriting (c), and its translation to SQL (d).

3. Translation of extensional FOFQ into SQL query.

We start with a formal definition of a faceted query tree.

Definition 5 (Faceted query tree). *A faceted query tree is an tree expression fqt conforming to the syntax:*

$$\begin{aligned} fqt &::= (root, qt) \\ qt &::= \circ\{ct_1, \cdots, ct_k\} \\ ct &::= (C, \varepsilon) \mid (C, \circ\{pt_1, \cdots, pt_m\}) \\ pt &::= (P, qt) \mid (P^-, qt) \mid (P, \circ\{vs_1, \cdots, vs_n\}) \\ vs &::= (a, \varepsilon) \mid (patt, \varepsilon), \end{aligned} \tag{1}$$

where: (a) $\circ \in \{\vee, \wedge\}$; (b) a pair $(lab, \circ\{t_1, \cdots, t_n\})$ denotes a tree with a root labeled by lab and a set $\{t_1, \cdots, t_n\}$ of subtrees; when $\circ$ is $\vee$, then the set of subtrees is treated as a disjunctive set, if $\circ$ is $\wedge$, then the set of subtrees is treated as a conjunctive set; (c) "root" is a distinguished label outside the ontology, $C \in \mathsf{UP}$, $P \in \mathsf{BP}$, $a \in \mathsf{Const}$, patt is a pattern denoting a set of constant values (strings) belonging to String.

Example 5. The faceted query tree in Fig. 3(b) is the following expression consistent with the syntax (1) (the *root* label will be always omitted):

$$\begin{aligned} T_1 = \vee\{(&Paper, \wedge\{(presentedAt, \vee\{(ACMConf, \varepsilon), (DEXAConf, \varepsilon)\}), \\ &(writtenBy, \vee\{(Person, \\ &\wedge\{(affiliation, \vee\{('INRIAFrance', \varepsilon), ('PUTPoland', \varepsilon)\})\})\})\})\} \end{aligned}$$

For the faceted query tree in Fig. 4(a), we have:

$$T_2 = \vee\{(ACMPaper, \wedge\{(paperTitle, \wedge\{(\%database\%, \varepsilon), (\%query\%, , \varepsilon)\})\})\}$$

The semantics of a faceted query tree is given by a first-order query (FOFQ), obtained by means of the semantic function $[\![\ ]\!]_x$, where x is a free variable in FOFQ.

Definition 6 (First order faceted query). *The semantics of a faceted query tree fqt, denoted $[\![fqt]\!]_x$ is the first-order query defined as follows:*

$$\begin{aligned}
[\![fqt]\!]_x &= [\![qt]\!]_x\\
[\![qt]\!]_x &= [\![ct_1]\!]_x \circ \cdots \circ [\![ct_k]\!]_x\\
[\![(C,\varepsilon)]\!]_x &= C(x)\\
[\![(C,\circ\{pt_1,\cdots,pt_m\})]\!]_x &= C(x) \wedge ([\![pt_1]\!]_x \circ \cdots \circ [\![pt_m]\!]_x)\\
[\![(P,qt)]\!]_x &= \exists y\ P(x,y) \wedge [\![qt]\!]_y\\
[\![(P^-,qt)]\!]_x &= \exists y\ P(y,x) \wedge [\![qt]\!]_y\\
[\![(P,\circ\{vs_1,\cdots,vs_n\})]\!]_x &= \exists y\ P(x,y) \wedge ([\![vs_1]\!]_y \circ \cdots \circ [\![vs_n]\!]_y)\\
[\![(a,\varepsilon)]\!]_x &= x = a\\
[\![(patt,\varepsilon)]\!]_x &= x \texttt{ LIKE } patt,
\end{aligned} \tag{2}$$

where the variable y under existential quantifier is "fresh:, i.e., any variable is at most once quantified.

In Fig. 3(c) and in Fig. 4(b), there are first-order faceted queries in forms of syntactic trees. Note, that those queries are built from both extensional and intensional predicates.

4.3 Transformation into Extensional First-Order Form and to SQL

Now, we show how a first-order faceted query is rewritten into an extensional form, i.e., a form, in which no intensional predicate appears. In the rewriting process, any atom built from an intensional predicate is rewritten according to the following procedure:

1. Let $P(y_1, y_2)$ be an atom in q, where P is an intensional predicate. Let $\omega = \alpha(x_1, x_2, x_3) \rightarrow P(x_1, x_2)$ be a rewriting rule of the form (RW-1). (RW-2) or (RW-3) in Table 1, and let variables x_1, x_2, x_3 do not appear in q.
2. The unification of $P(y_1, y_2)$ and $P(x_1, x_2)$, implies the substitution $\theta = [x_1/y_1, x_2/y_2]$, i.e., any occurrence of x_i must be substituted by y_i, $i = 1, 2$.
3. The atom $P(y_1, y_2)$ is replaced in q with $\alpha(x_1, x_2, x_3)[x_1/y_1, x_2/y_2]$, i.e., with the body of the rewriting rule with variables substituted accordingly.
4. A similar procedure is applied for any atom $C(y)$, where C is an intensional predicate, and the rewriting rule is of the form (RW-3) or (RW-4).

Example 6. Let us consider the following query, that is a fragment of FOFQ in Fig. 3(c):

$$q(x) = Paper(x) \wedge \exists x_1\ presentedAt(x, x_1) \wedge ACMConf(x_1).$$

To rewrite the atom $presentedAt(x, x_1)$, we can use the following rewriting rule of the form (RW-1), given in Example 1(4), in which names of variables are changed to be different from those in the considered FOFQ in Fig. 3(c):

$$\exists x_4\ inProc(x_5, x_4) \wedge Proceedings(x_4) \wedge ofConf(x_4, x_6) \rightarrow presentedAt(x_5, x_6).$$

Then, the substitution unifying the considered atom and the head of the rewriting rule, is $\theta = [x_5/x, x_6/x_1]$. Thus, after rewriting we have

$$q_1(x) = Paper(x) \wedge \exists x_4\ inProc(x, x_4) \wedge Proceedings(x_4) \wedge \exists x_1\ ofConf(x_4, x_1) \\ \wedge ACMConf(x_1).$$

Extensional forms of the considered faceted queries are presented in Figs. 3(d) and 4(c), respectively. Finally, the extensional FOFQ is translated into SQL query. For our running examples, results of these translations are shown in Fig. 5, and in Fig. 4(d), respectively.

```
SELECT DISTINCT x.Id FROM Paper as x
JOIN Proceedings as x4 ON x.ProcId = x4.Id
JOIN Conference as x1 ON x4.ConfId = x1.Id AND x1.Acronym LIKE '%ACM%'
JOIN AuthorPaper as x_x2 ON x.Id = x_x2.PaperId
JOIN Person as x2 ON x_x2.PersonId = x2.Id
JOIN Affiliation as x2_x3 ON x2.Id = x2_x3.PersonId
AND x2_x3.Affiliation IN ('INRIA, France','PUT, Poland')
UNION SELECT DISTINCT x.Id FROM Paper as x
JOIN Proceedings as x4 ON x.ProcId = x4.Id
JOIN Conference as x1 ON x4.ConfId = x1.Id AND x1.Acronym LIKE '%DEXA%'
JOIN AuthorPaper as x_x2 ON x.Id = x_x2.PaperId
JOIN Person as x2 ON x_x2.PersonId = x2.Id
JOIN Affiliation as x2_x3 ON x2.Id = x2_x3.PersonId
AND x2_x3.Affiliation IN ('INRIA, France','PUT, Poland')
```

Fig. 5. SQL query created from extensional FOFQ in Fig. 3(d).

5 Summary

In this paper, we presented solutions to some model and language transformations in ontological databases. We considered an ontological database as a knowledge base, where its schema (ontological schema) is defined in a form of an ontology, and the extensional part is stored in a relational database. We studied two transformation problems in such scenario: (a) a mapping from the terminological part of the ontology into relational database schema (a meta-schema mapping), and (b) a mapping from an extensional part of the ontology into relational database instance (a schema mapping). We focused also on transformations connected with formulating faceted queries over an ontological schema. A faceted query is created using a faceted interface, and must be: (a) transformed into a first-order query, (b) rewritten into extensional first-order query, and finally (c) translated into SQL query, that is executed in relational database. The considerations are illustrated by solutions implemented in the DAFO (*Data Access based in Faceted queries over Ontologies*) system [14,17].

This research has been supported by Polish Ministry of Science and Higher Education under grant 04/45/DSPB/0185.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
2. Arenas, M., Grau, B.C., Kharlamov, E., Marciuska, S., Zheleznyakov, D.: Faceted search over ontology-enhanced RDF data. In: ACM CIKM 2014, pp. 939–948 (2014)
3. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Petel-Schneider, P. (eds.): The Description Logic Handbook: Theory Implementation and Applications. Cambridge University Press, New York (2003)
4. Calì, A., Gottlob, G., Lukasiewicz, T., Pieris, A.: $Datalog+/-$: a family of languages for ontology querying. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2.0 2010. LNCS, vol. 6702, pp. 351–368. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24206-9_20
5. Calì, A., Gottlob, G., Pieris, A.: Advanced processing for ontological queries. PVLDB **3**(1), 554–565 (2010)
6. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rosati, R.: Ontology-based database access. In: SEBD 2007, pp. 324–331 (2007)
7. Calvanese, D., Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: the DL-Lite family. J. Autom. Reason. **39**(3), 385–429 (2007)
8. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. Theor. Comput. Sci. **336**(1), 89–124 (2005)
9. Fagin, R., Kolaitis, P.G., Popa, L.: Data exchange: getting to the core. ACM Trans. Database Syst. **30**(1), 174–210 (2005)
10. Gottlob, G., Orsi, G., Pieris, A.: Query rewriting and optimization for ontological databases. ACM Trans. Database Syst. **39**(3), 25:1–25:46 (2014)
11. Gruber, T.R.: Toward principles for the design of ontologies used for knowledge sharing? Int. J. Hum. Comput. Stud. **43**(5–6), 907–928 (1995)
12. Hahn, R., Bizer, C., Sahnwaldt, C., Herta, C., Robinson, S., Bürgle, M., Düwiger, H., Scheel, U.: Faceted Wikipedia search. In: Abramowicz, W., Tolksdorf, R. (eds.) BIS 2010. LNBIP, vol. 47, pp. 1–11. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12814-1_1
13. Motik, B., Horrocks, I., Sattler, U.: Bridging the gap between OWL and relational databases. J. Web Semantics **7**(2), 74–89 (2009)
14. Pankowski, T.: Exploring ontology-enhanced bibliography databases using faceted search. In: Kamps, J., Tsakonas, G., Manolopoulos, Y., Iliadis, L., Karydis, I. (eds.) TPDL 2017. LNCS, vol. 10450, pp. 27–39. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67008-9_3
15. Pankowski, T.: Rewriting and executing faceted queries over ontology-enhanced databases. In: 21st Conference on Knowledge-Based and Intelligent Systems (KES 2017), pp. 137–146. Procedia Computer Science, Elsevier (2017)
16. Pankowski, T., Brzykcy, G.: Data access based on faceted queries over ontologies. In: Hartmann, S., Ma, H. (eds.) DEXA 2016. LNCS, vol. 9828, pp. 275–286. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44406-2_21
17. Pankowski, T., Brzykcy, G.: Faceted query answering in a multiagent system of ontology-enhanced databases. In: Jezic, G., Chen-Burger, Y.-H.J., Howlett, R.J., Jain, L.C. (eds.) Agent and Multi-Agent Systems: Technology and Applications. SIST, vol. 58, pp. 3–13. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39883-9_1

18. Reiter, R.: On closed world data bases. In: Logic and Data Bases, pp. 55–76 (1977)
19. Sequeda, J.F., Arenas, M., Miranker, D.P.: OBDA: query rewriting or materialization? In practice, both!. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 535–551. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_34
20. Sherkhonov, E., Cuenca Grau, B., Kharlamov, E., Kostylev, E.V.: Semantic faceted search with aggregation and recursion. In: d'Amato, C., Fernandez, M., Tamma, V., Lecue, F., Cudré-Mauroux, P., Sequeda, J., Lange, C., Heflin, J. (eds.) ISWC 2017. LNCS, vol. 10587, pp. 594–610. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68288-4_35
21. Skjæveland, M.G., Giese, M., Hovland, D., Lian, E.H., Waaler, A.: Engineering ontology-based access to real-world data sources. J. Web Sem. **33**, 112–140 (2015)
22. SPARQL Query Language for RDF (2008). http://www.w3.org/TR/rdf-sparql-query
23. TreeView Class (2017). https://msdn.microsoft.com/en-us/library/system.windows.forms.treeview(v=vs.110).aspx
24. Tunkelang, D.: Faceted Search. Morgan & Claypool Publishers, San Rafael (2009)
25. Tzitzikas, Y., Manolis, N., Papadakos, P.: Faceted exploration of RDF/S datasets: a survey. J. Intell. Inf. Syst. **48**, 1–36 (2016)
26. Vega-Gorgojo, G., Slaughter, L., Giese, M., Heggestøyl, S., Soylu, A., Waaler, A.: Visual query interfaces for semantic datasets: an evaluation study. J. Web Semantics **39**, 81–96 (2016)
27. Wagner, A., Ladwig, G., Tran, T.: Browsing-oriented semantic faceted search. In: Hameurlain, A., Liddle, S.W., Schewe, K.-D., Zhou, X. (eds.) DEXA 2011. LNCS, vol. 6860, pp. 303–319. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23088-2_22

Model Transformation Reuse Across Metamodels
A Classification and Comparison of Approaches

Jean-Michel Bruel[1], Benoit Combemale[1], Esther Guerra[2](✉), Jean-Marc Jézéquel[3], Jörg Kienzle[4], Juan de Lara[2], Gunter Mussbacher[4], Eugene Syriani[5], and Hans Vangheluwe[4,6]

[1] University of Toulouse, IRIT, Toulouse, France
[2] Universidad Autónoma de Madrid, Madrid, Spain
esther.guerra@uam.es
[3] Univ Rennes, Inria, CNRS, IRISA, Rennes, France
[4] McGill University, Quebec, Canada
[5] Université de Montréal, Montréal, Canada
[6] University of Antwerp, Antwerp, Belgium

Abstract. Model transformations (MTs) are essential elements of model-driven engineering (MDE) solutions. MDE promotes the creation of domain-specific metamodels, but without proper reuse mechanisms, MTs need to be developed from scratch for each new metamodel. In this paper, we classify reuse approaches for MTs across different metamodels and compare a sample of specific approaches – model types, concepts, a-posteriori typing, multilevel modeling, and design patterns for MTs – with the help of a feature model developed for this purpose, as well as a common example. We discuss strengths and weaknesses of each approach, provide a reading grid used to compare their features, and identify gaps in current reuse approaches.

Keywords: Model transformation · Reuse · Classification
Feature model · Model types · Concepts · A-posteriori typing
Multilevel modeling · Transformation design patterns

1 Introduction

As model-driven engineering (MDE) is used for engineering evermore numerous and complex systems, model transformations (MTs) are becoming more and more complex pieces of software. Like for any other piece of software [1], *reuse mechanisms* for MTs have been proposed to limit reimplementing a transformation from scratch every time a new but related need arises. In this paper, we focus on the reuse of MTs that were developed for a particular metamodel, but are then applied to models typed by other metamodels, i.e., reuse *across* metamodels.

A. Rensink and J. Sánchez Cuadrado (Eds.): ICMT 2018, LNCS 10888, pp. 92–109, 2018.
https://doi.org/10.1007/978-3-319-93317-7_4

Many use cases of MT reuse have been identified in the literature [2], providing useful classifications. Since the use cases of MT reuse imply very different trade-offs among non-functional properties such as type-safety, performance, expressiveness and user-friendliness, no single MT reuse approach fits them all.

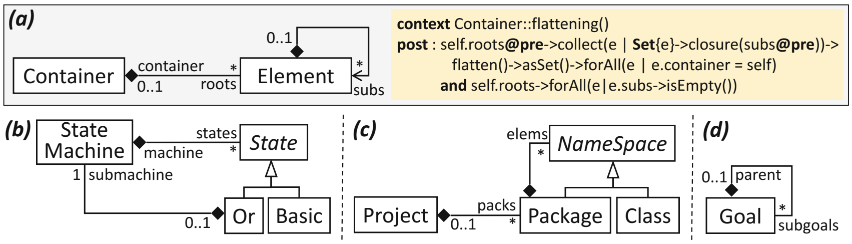
In this paper, we propose a classification of MT reuse approaches that work across metamodels, and compare a sample of specific approaches—namely model types [3,4], concepts [5,6], a-posteriori typing [7], multilevel modeling [8], and design patterns for MTs [9]—with the help of a feature model developed for this purpose, and a common example. We discuss strengths and weaknesses of each proposal, provide a reading grid to compare their features, and identify gaps in current reuse approaches.

The paper is organized as follows. Section 2 motivates the need for reuse mechanisms across metamodels and presents a running example. Section 3 defines classification criteria using a feature model. Section 4 compares five existing approaches based on the classification and the running example, and Sect. 5 discusses trade-offs. Section 6 overviews related classification attempts and reuse techniques, and Sect. 7 concludes by identifying challenges for the MT community.

2 Motivation

MDE supports the creation of metamodels to describe models using the most appropriate primitives and level of abstraction. However, this entails the creation of all kinds of services for each metamodel, including MTs. Without proper reuse mechanisms, MTs need to be created from scratch even if there are MTs with the same goal but defined over similar yet different metamodels.

As a concrete example, consider a MT that implements the common *flattening* operation. This MT traverses a given hierarchy and extracts its elements into a flat collection. Figure 1(a) illustrates a specification for such a MT, defined over a minimal metamodel that contains just the elements the MT needs (Container and Element). In practice, the MT would be implemented using languages like ATL [10], ETL [11], or Kermeta [12], but to stay language-agnostic, we only show a post-condition that identifies its effect. The first two lines of the postcondition state that, for a given hierarchy, all (sub-)elements should become contained in the same root container; the last line ensures the hierarchy is removed.



```
context Container::flattening()
post : self.roots@pre->collect(e | Set{e}->closure(subs@pre))->
        flatten()->asSet()->forAll(e | e.container = self)
    and self.roots->forAll(e|e.subs->isEmpty())
```

Fig. 1. (a) Reusable model transformation scheme. (b, c, d) Metamodels for which the model transformation wants to be reused.

Flattening is recurrent in many contexts, like in structural modeling (class/package hierarchies, goal hierarchies) and behavioral languages (state machines, activity diagrams). Figures 1(b), (c), (d) show three typical metamodels of these kinds of languages.

Without proper reuse mechanisms, a flattening MT needs to be implemented from scratch for each metamodel. Some ad-hoc reuse approaches are applied in practice, like *clone-and-own* (copy-paste and manual adaptation) or translating the models of interest to the metamodel accepted by the reused MT. Neither approaches are optimal. In the first case, manual adaptation is time-consuming, error-prone, hardly scalable, and leads to well-known maintenance problems with code clones [13]. In the second case, illustrated by Fig. 2, an existing MT (*rt* on the right) defined for a metamodel MM, wants to be reused on a model (M' on the left) conformant to a different metamodel MM'. In this figure (and following ones), light boxes represent existing artifacts, and dark ones represent new artifacts to be built. An *adapter* transformation is required to translate the model into one that conforms to the metamodel the reused transformation conforms to, so that the MT can be applied to this new model M. This is not efficient since it requires executing an additional transformation in addition to the reused one. Moreover, a reverse MT may be needed to transform the result back to the original model's metamodel.

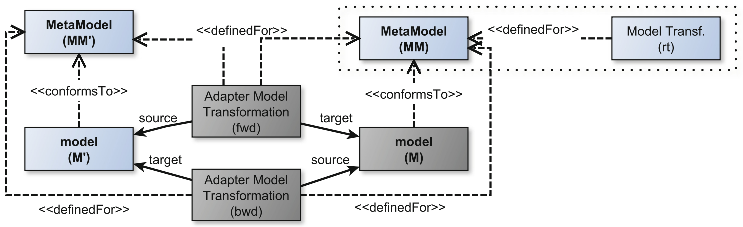


Fig. 2. Explicit model adaptation approach to MT reuse

The MT community has proposed several approaches to facilitate reuse across metamodels, like model typing, a-posteriori typing, concepts, multilevel modeling and transformation patterns, among others [14–17]. These approaches have different trade-offs and are applicable in different scenarios and contexts. Hence, there is an urging need to classify and compare them to know which approach to use in a given situation.

3 Classification

We introduce a feature model to classify the different alternatives for MT reuse across metamodels. The model, shown in Figs. 3 and 4, presents the features of the reuse mechanism as well as properties of the reused transformation. In the following, we write rt to denote the MT to be reused.

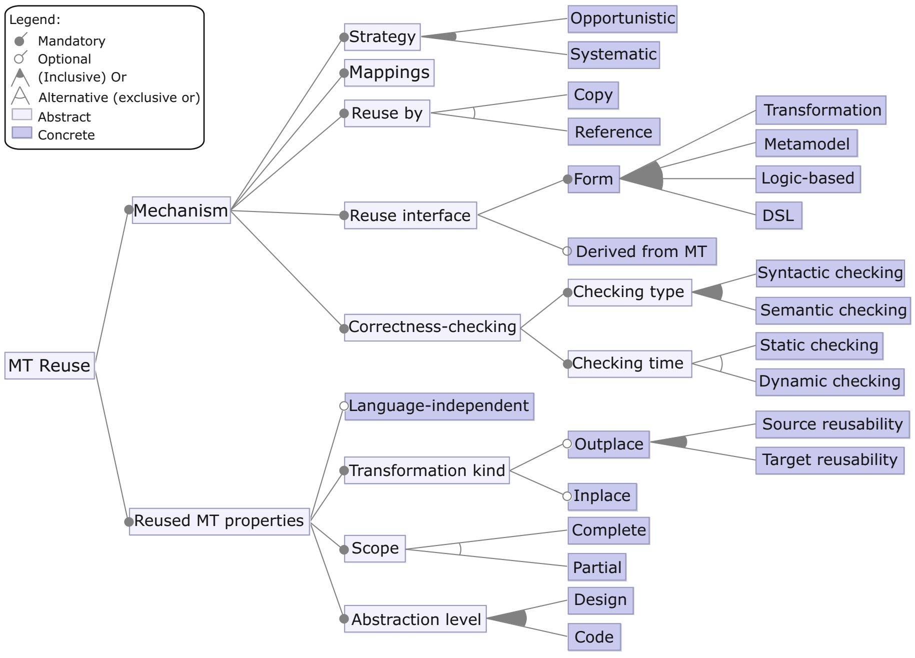


Fig. 3. Feature model: mechanisms for reuse and scenarios of reuse (the *Mapping* feature is expanded in Fig. 4

Strategy. In a *systematic* reuse strategy, a MT is developed by reusing specific units that were made available a priori. This is analogous to software built following a component-based design. In this case, rt was developed with the intention of being reused. Hence, depending on the reuse approach, the MT needs to be packaged as a component [6], as a pattern [9], or the metamodel the MT is defined on needs to be sliced [18]. All other kinds of reuse are considered *opportunistic*.

Mappings. A reusable transformation rt, defined over a metamodel MM, is applicable to a number of different metamodels MM'. The way to specify the correspondences or mappings between MM and MM' depends on the reuse approach, and determines the set of metamodels where rt can be reused. Figure 4 shows the alternative features for mapping specifications.

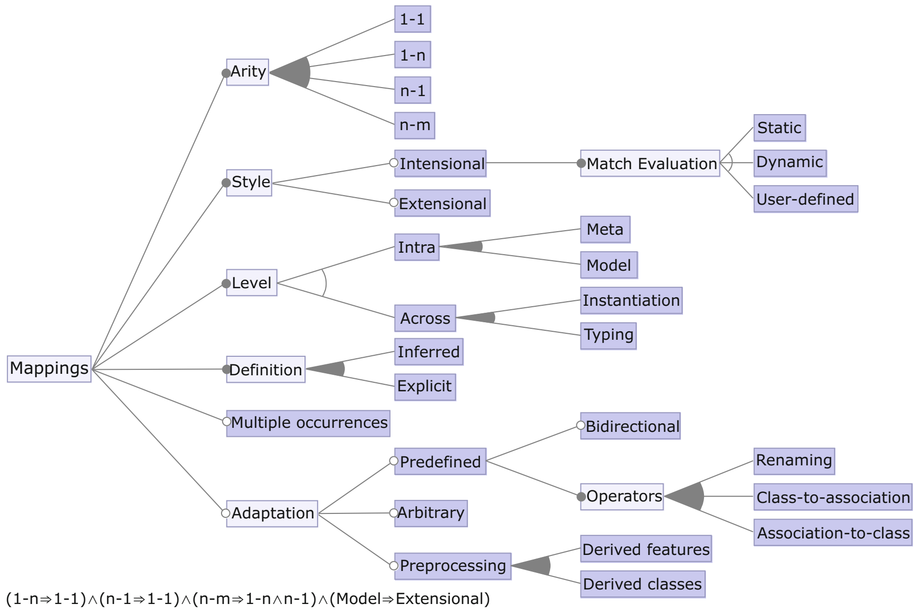


Fig. 4. Feature model: specification of mappings

- **Arity:** The relation between MM and the new reuse context MM' can be *one-to-one*: injective where each element in MM needs to be mapped to exactly one element in MM'. The mapping can be *one-to-many*: each MM element is mapped to any number of MM' elements, including none. It can also be *many-to-one*: an MM element can be mapped several times. Finally, the most general kind of mapping is *many-to-many*: elements in both MM and MM' can be mapped several times.
- **Style:** The objects over which rt are reused can be specified either by *extension* (i.e., enumerating them) or by *intension* (i.e., providing necessary and sufficient conditions that characterize the objects). Moreover, intensional specifications can be evaluated statically at *compile-time*, *dynamically* at run-time, or at the convenience of the user (*user-defined*).
- **Level:** *Intra-level* mappings relate elements at the same metalevel: either two *meta*models, which is the most common case, or two *models*. In contrast, mappings *across* levels relate elements at different metalevels by means of *instantiation* (e.g., in multilevel modeling) or *typing* relationships (e.g., in transformation patterns, where rule elements are typed w.r.t. a metamodel).
- **Definition:** The mapping between MM and MM' can be *explicit*, i.e., defined by the user (using either an extensional or intensional approach), or be *inferred* automatically, e.g., using name matching [3] or structural similarity criteria [14].
- **Multiple Occurrences:** This refers to the possibility to define multiple application contexts for rt within a metamodel MM', all of which are handled

simultaneously by *rt*, perhaps using a composition mechanism for coordination. Most existing approaches only support one application context at a time.

– **Adaptation:** To widen the number of metamodels where a transformation can be reused, several mechanisms bridge heterogeneities between MM and MM'. Some approaches provide a set of *predefined* operators for specific kinds of adaptations, such as *renaming* a class, mapping a *class to* an *association*, or mapping an *association to* a *class* [6,14] (please note that our feature model does not list all possible predefined adaptation operators). Such operators may be *bidirectional* or not. Other approaches allow *arbitrary* adaptations between MM and MM', usually defined by means of OCL expressions. It is also possible to rely on a *preprocessing* step that adds the necessary *derived classes* or *derived features* to MM', making it structurally similar to MM and allowing a direct mapping between them, before applying *rt* [6,19].

Reuse By. This feature refers to whether the original transformation is *copied* or *referenced*. In the *clone-and-own* approach (cf. Sect. 2), the developer reuses a copy of *rt* in the transformation. Therefore, any updates to *rt* will not be propagated to the new transformation. Instead, the *adapter* approach of Fig. 2 reuses *rt* by reference, and hence any further update to the transformation affects all places where it was reused.

Reuse Interface. Reusable transformations expose an interface for reuse that can take different *forms* depending on the approach. It can be a *metamodel* declaring the necessary classes and features in the context of reuse [3,4,6,9], a *logic-based* specification stating the constraints that a metamodel should fulfill to ensure a correct MT reuse [17], or a model describing metamodel requirements using a domain-specific language (*DSL*) [15]. Sometimes, this reuse interface can be (semi-)automatically *derived from the MT* [15,17,18]. While the above-mentioned interface kinds yield a black-box approach to reuse, the interface for reuse in white-box approaches is the reusable MT or an abstraction of it [9,14]. This is appropriate when a larger MT is to be composed out of smaller fragments. Both interface kinds can be combined.

Correctness Checking. Different approaches make different choices on how and when the correctness of *rt* with respect to m' and MM' should be checked.

– **Checking-Type:** Checking can be either *syntactic*, e.g., simple type checking, or *semantic*, typically also verifying the satisfaction of well-formedness rules expressed in OCL, or additional semantic conditions capturing the transformation *intent* (e.g., like bisimulation relations) [20].

– **Checking-Time:** When the correctness of *rt* is checked *statically*, it is ensured that it will be syntactically correct for all models conforming to the new context of reuse MM'. Instead, a *dynamic* check needs to inspect at run-time that every (read/write) access to the model by *rt* is correct. Static checking of semantic properties requires some form of theorem proving or model checking, while dynamic checking only requires a run-time evaluation of OCL constraints.

Properties of Reused Transformation. Transformation reuse approaches can be *language-independent* (i.e., the reusable transformation can be written in any transformation language) or be specific for a transformation language (e.g., ATL or graph transformation). Moreover, some approaches may be limited to a particular kind of transformation, application scope or abstraction level.

– **Transformation Kind:** The reused transformation can be either *inplace* or *outplace* (i.e., model-to-model). In the former case, the mechanism needs to ensure that write accesses to the model are correct. In the latter case, the new context of reuse can be for the source metamodel, which is typically read-only (*source reusability*), for the target metamodel, which is typically write-only (*target reusability*), or for both.
– **Scope:** The reused unit can be a *complete* model transformation or a part of it, e.g., a rule (*partial*).
– **Abstraction Level:** Reuse can be at the *design* level, e.g., in the form of design patterns [9], or directly at the implementation level to reuse transformation *code*.

4 Comparison of Some Existing Approaches

In this section, we analyze five prominent reuse approaches, classifying them by the introduced feature model. Each approach is based on a different technique, summarized in Fig. 5. Model types (Fig. 5a) is based on establishing a subtyping relation between metamodels. A-posteriori typing (Fig. 5b) works by retyping the model so that the reused MT can be applied to it. Concepts rely on genericity to rewrite the MT using a high-order transformation (Fig. 5c) to make it applicable to a particular metamodel. Similarly, MT patterns use a generative approach to

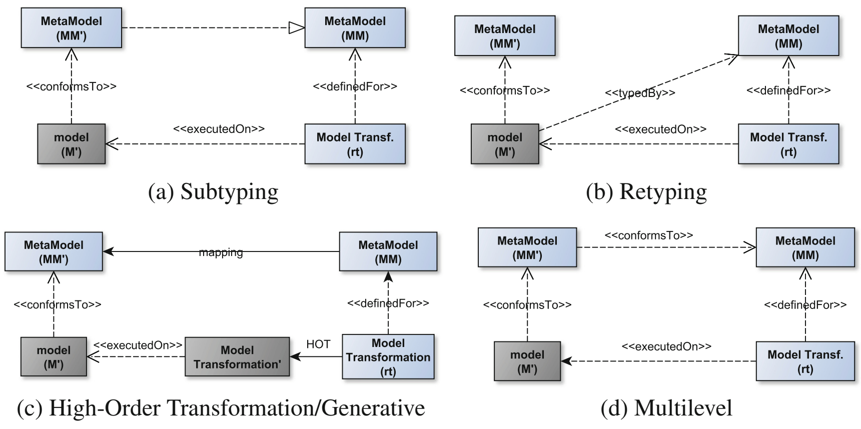


Fig. 5. Different techniques enabling MT reuse across metamodels

synthesize specific MT code from a design pattern. Finally, multilevel modeling exploits the typing relation to apply the MT two (or more) metalevels below (Fig. 5d).

Table 1 summarizes how each approach instantiates the feature model. We provide more details on their working scheme using the running example in what follows.

4.1 Model Typing

Model Types were introduced by Steel et al. [3], as an extension of object typing to provide abstraction from object types and enable model manipulation reuse. The type of a model is a set of types of objects that may belong to the model, and their relationships. While a model conforms to one and only one metamodel (the one containing all the types needed to instantiate objects of the model), it can have several model types which are subsets of its metamodel. Substitutability is the ability to safely use a model of type A where a model of type B is expected. Substitutability is supported in the model type theory by defining a subtyping relationship among model types [4,21,22].

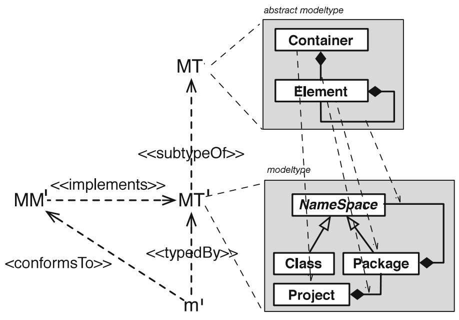


Fig. 6. Reuse with model typing

Figure 6 illustrates model typing, showing how to reuse the flattening MT defined on MT for the object-oriented metamodel MM'. Based on derived attributes defined within the object-oriented metamodel, if an isomorphism is statically (or possibly) found, the flattening MT can be safely applied on the instances of the object-oriented metamodel (m'). Melange employs adapter generators at compile time to ensure the adaptation at runtime of the actual application of the MT on the instances of the targeted metamodel [22].

4.2 Concepts

Inspired by generic programming, concepts were proposed in [5] as a mechanism to express requirements for generic model management operations and transformations. A concept is similar to a metamodel, but its elements are parametric types that need to be bound to elements in a metamodel. Generic transformations are defined over concepts. When a concept is bound to a metamodel, the associated transformation gets rewritten in terms of the metamodel and can be applied to its instances. In this approach, adapters [6] enable more flexible bindings by the use of OCL expressions in mappings, which get injected in the rewritten MT code.

Figure 7 shows how to reuse the flattening MT for the object-oriented metamodel using concepts. The flattening metamodel is considered the concept, whose elements need to be bound to elements in the concrete metamodel. In this case,

Table 1. Classification of MT reuse approaches

Feature	Model-typing	Concepts	A-posteriori	Multilevel	MT patterns
Mechanism					
Strategy	Systematic opportunistic	Systematic opportunistic	Systematic opportunistic	Systematic opportunistic	Systematic
Reuse by	Reference	Copy	Reference	Reference	Copy
Reuse interface	Metamodel can be derived	Metamodel can be derived	Metamodel	Metamodel	Transformation
Checking type	Syntactic semantic (pre. and post.)	Syntactic	Syntactic	Syntactic	Syntactic
Checking time	Static (type-level) Dynamic (inst-level)	Static	Static (type-level) Dynamic (inst-level)	Static	Static
Mechanism.Mappings					
Arity	1−1, 1−n, n−1, n−m[a]	1−1, 1−n, n−1, n−m	1−1, 1−n, n−1, n−m	1−1, 1−n	1−1
Style	Extensional	Extensional	Extens. (type-level) Intens. (inst-level) Dynamic match	Extensional	Extensional
Level	Intra/meta	Intra/meta	Intra/meta	Across/ instantiation	Across/typing
Definition	Inferred	Explicit	Explicit	Explicit	Explicit
Multiple occur.	No	No	No	No	No
Adaptation	Renaming, derived feats	Renaming, c-to-a a-to-c, arbitrary, derived feats, derived classes	Renaming, arbitrary, bidirectional, derived feats	Renaming, derived feats	Renaming
Reused MT properties					
Lang. indep.	Yes	No	Yes	Yes	Yes[b]
Transf. kind	Inplace Outplace (M2M & M2T)	Inplace [5] Outplace [6] src/tar reusability	Inplace Outplace src/tar reusability	Inplace Outplace src/tar reusability	Inplace Outplace src/tar reusability
Scope	Complete	Complete	Complete	Partial[c]	Partial
Abstrac. level	Code	Code	Code	Code	Design

[a]Preprocessing of derived features for alignment
[b]By additional code generators
[c]Through refining transformations [5]

an adapter is needed to filter Class objects out of the elems relation (see last line of binding). As a last step, the generic transformation is rewritten using the bindings and the adapters.

4.3 A-posteriori Typing

A-posteriori typing [7] permits classifying objects by classes different from the ones used to initially create the objects, and hence enables multiple, partial,

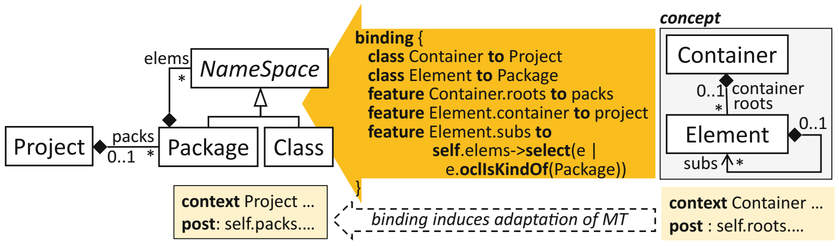


Fig. 7. Binding of concept to metamodel, and MT adaptation (sketch)

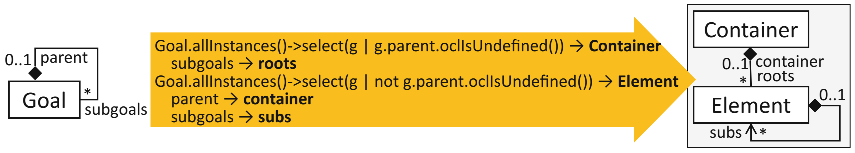

0..1 parent
Goal
* subgoals
Goal.allInstances()->select(g | g.parent.oclIsUndefined()) → Container
subgoals → roots
Goal.allInstances()->select(g | not g.parent.oclIsUndefined()) → Element
parent → container
subgoals → subs
Container
0..1 container
* roots
Element
0..1
subs *

Fig. 8. A-posteriori instance-level specification for the flattening of goal models

dynamic typings. This approach allows opportunistic reuse as MTs defined for a metamodel can be reused with other models after being reclassified. In this way, MTs become highly reusable as, similar to Java interfaces, one can design metamodels whose goal is not object creation, but to serve as a type for MTs. Figure 5b shows the working scheme of this approach: a model typed by an arbitrary metamodel is assigned new types from the metamodel a MT was defined on, and as a result, the MT can be executed as-is on the model.

A-posteriori typing specifications can be type-level or instance-level. The former induces a static relation between two metamodels, so that instances of one can be seen as instances of the other. This mapping style is similar to those in model typing. Instance-level specifications are more expressive than type-level ones, as they permit classifying objects by queries that assign a given type to the result of the query. This typing is dynamic because classification may depend on the run-time value of the object properties, which may evolve. Moreover, it allows an object to have multiple a-posteriori types.

Figure 8 shows an instance-level a-posteriori specification to reuse the flattening transformation with goal models. In particular, all Goal objects with no parent are retyped as Containers, all Goal objects with a parent goal are retyped as Elements, and references are also retyped properly. When a goal model gets retyped by this specification, the MT can be applied as-is on the model. This instance-level example that partitions Goal objects into two sets at run-time illustrates the power of dynamic match evaluation, which among the surveyed approaches is only supported by a-posteriori typing.

4.4 Multilevel Modeling

Multilevel modeling was proposed in [23] as a way to enhance flexibility in modeling by enabling an arbitrary number of metalevels and a dual type/instance facet for model elements, so that they are instances with respect to the metalevel above, and types with respect to the metalevel below. This approach facilitates the definition of domain-specific metamodeling languages and families of languages [8], which can be iteratively refined in successive metalevels to account for domain-specific aspects. Model management operations defined in upper metalevels become generic and applicable to the instances in direct and indirect lower metalevels.

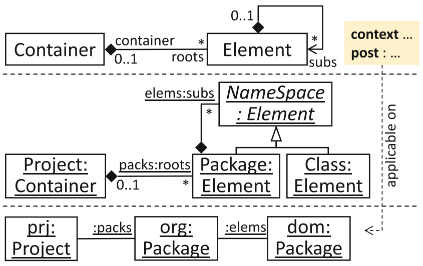


Fig. 9. Reuse by multilevel modeling

Figure 9 uses multilevel modeling to reuse the flattening transformation with a metamodel for object-oriented design. The metamodel of the flattening transformation needs to be promoted to a higher metalevel, and the object-oriented design metamodel needs to be created as an instance of it. In this way, the transformation can be applied on the object-oriented models created in the lower metalevel.

4.5 Design Patterns for Model Transformations

Design patterns are artifacts reputed for reuse in software engineering. Unlike the previous approaches, reuse must be planned for at design-time. The approach in [9] introduces a DSL, called DelTa, to define design patterns for MTs. Given a pattern in DelTa, a higher-order transformation (HOT) synthesizes a partial MT that implements the pattern in a dedicated MT language by means of code generation. A DelTa model describes an ordered set of rules containing abstract entities and relations that can be matched (positively or negatively), created, or deleted.

The top of Fig. 10 shows a design pattern in DelTa representing the flattening operation that satisfies the specification in Fig. 1. It consists of three rules that must be applied in this order on a given metamodel mm. It is thus an inplace transformation. The roots rule creates a trace link (dotted arrow) from the container to the root elements and removes the roots relation. In DelTa notation, elements in gray shall be created, those in black shall be removed, and all others are part of the constraint that shall be matched. Elements labeled with n0 are part of the negative constraint that shall not be matched. The closure rule creates a trace link from the container to all sub-elements recursively (i.e., the transitive closure). The leaves rule creates a roots relation from the container to all elements with no sub-element. The Flatten design pattern and the mapping

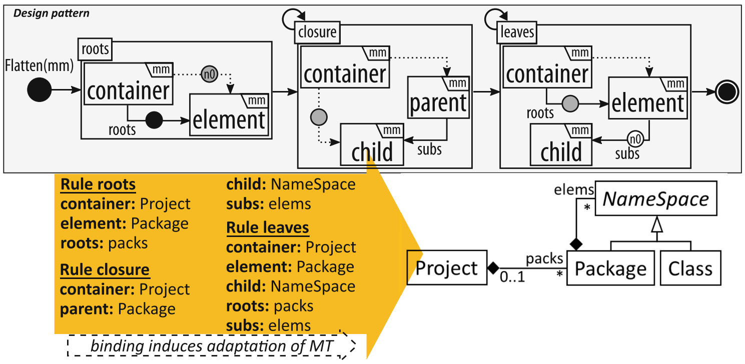


Fig. 10. Binding of flattening design pattern to metamodel

are specified independently from the MT language. However, the HOT generates its implementation in a specific MT language for a specific metamodel.

Using the notation in Fig. 5c for the MT patterns approach, MM corresponds to the metamodel of DelTa (see [9]), rt is the Flatten design pattern, and MM' is the object-oriented design metamodel in this example. Then, similar to the concepts approach, rt is reused by generating a MT tailored to MM'.

5 Discussion

From the configurations shown in Table 1 for several MT reuse approaches, next, we discuss their differences with regards to a number of properties: if reuse is opportunistic or systematic, the customization techniques used to adapt a MT to a particular context, the customization ease and expressiveness, the overhead at execution time, and the properties guaranteed by the approaches. Table 2 synthesizes the results.

To reuse a MT, it is first necessary to make it reusable. This can be done a priori when the MT is defined (i.e., systematic reuse) or a posteriori when the MT is reused (i.e., opportunistic reuse). Model typing, concepts, a-posteriori typing and multilevel modeling support both kinds of reuse. For opportunistic reuse, the former two provide slicing mechanisms to extract the relevant part of the metamodel used by the MT [18], and for planned reuse, they support the definition of the MT on a generic metamodel (called *abstract* in model typing and *concept* in the concepts approach) which is the minimal metamodel the MT requires. Multilevel modeling uses promotion (i.e., pulls a metamodel one metalevel up) to handle opportunistic reuse, and it creates deep metamodels (i.e., which can be instantiated in successive metalevels) for systematic reuse. In a-posteriori typing, there is no specific technique to simplify opportunistic reuse, while for systematic reuse one can create a role metamodel [7] (i.e., its primary goal is not instantiation but retyping). Patterns are only relevant for systematic reuse, where abstract patterns are made available to be applied on a specific metamodel.

Table 2. Comparison of model transformation reuse approaches

	Model-typing	Concepts	A-posteriori	Multilevel	MT patterns
Reusing existing MT (opportunistic)	Slicing	Slicing	Free	Promotion	N.A.
Making a MT reusable (systematic)	Abstract MM	Generic MM (concept)	Role MM	Deep MM	Design pattern
Customization technique	Adapter + Mappings	Adapter + Mappings	Adapter + Mappings	Instantiation	Mappings
Customization ease	Low to high	Low to high	Low to high	Medium to high	High
Customization expressiveness	High (polymorphic reuse)	Medium-high (parametric reuse)	Very high (multi-matching, dynamic)	Medium (instantiation)	Low (limited matching)
Execution cost overhead	Evaluation of adapter at run-time	None (adapter injected in MT at compile time)	Evaluation of adapter at run-time	Traverse typing relations at run-time	None (MT excerpt generated from pattern)
Property preservation guarantees	Static typing, polymorphism	Static typing, generics	Dynamic typing, constraint solving	Static typing, multilevel	Static typing, generative

Once the MT rt is available for reuse, it is necessary to align the initial metamodel MM over which it is defined, to the actual metamodel MM' on which it is to be reused. Model typing, concepts, a-posteriori typing, and patterns rely on syntactic mappings. When further customizations are required to apply rt in a particular context, model typing, concepts, and a-posteriori typing also support the definition of explicit adapters. Multilevel modeling relies on instantiation to map the initial metamodel MM to the actual metamodel MM' one metalevel below. In the case of patterns, the developer must typically refine the MT by hand if the mapping is complex.

The complexity of the adapters depends on the syntactic distance between the initial and actual metamodels. The cost to specify them can range from low to high accordingly. Multilevel modeling requires a special metamodeling architecture, and patterns require an explicit definition of the mapping even in case of an isomorphic alignment, while other approaches may infer it automatically.

Regarding the expressiveness of the mapping customization, model-typing relies on polymorphic reuse and concepts on parametric reuse. A-posteriori typing supports in addition multi-matching (i.e., a model element can get several a-posteriori types) and dynamic typing. Multilevel modeling uses instantiation for customization, and patterns are limited to isomorphic matching.

The expressiveness for defining the customization comes with the cost of its evaluation when the MT is reused. Model typing and a-posteriori typing evaluate the adapters when the MT is called, and multilevel modeling follows a similar approach by traversing the typing relationships at run-time. However, the added flexibility of a-posteriori typing for instance-level specifications may incur run-time penalties, as object types are dynamically calculated by queries. The concepts approach evaluates the adapters at compile-time to generate a new MT fitting the new metamodel MM'. The execution cost is not applicable for patterns since they are reused at design-time [9], and then compiled into a specific MT language.

Finally, the property preservation guarantee relies on the underlying theory used by each approach. At design-time, model typing relies on polymorphic reuse, concepts rely on parametric reuse, multilevel modeling relies on deep instantiation, and patterns use a generative approach. A-posteriori typing uses constraint solving at design-time to discard potentially unsafe matchings, but the correctness guarantees are limited by the bounded search of the constraint solver [7].

Altogether, the discussed approaches cover most features in the feature model, but a few remain uncovered. Two specification styles are not favored by any approach. First, with respect to intensional specification of mappings, they are either evaluated statically (in model types) or dynamically (in a-posteriori typing); however, having user-defined evaluation points in the transformation execution is unexplored. As for the level of mappings, they are either across levels (instantiation for multilevel modeling, and typing for patterns) or intra-level between metamodels (the rest); however, no approach supports intra-level mappings between models. This latter specification style could be realized by mapping the model elements to be transformed to the elements in reused rules, which would lead to highly customized but very costly reuse specifications.

Other uncovered options relate to the functionality offered by the reuse mechanism. First, supporting semantic checkings (i.e., in line with the so-called transformation "intents" [17,24]) would be a way to further characterize correct reuse contexts by expressing requirements on the expected (possibly dynamic) semantics of the reuse context. To our knowledge, there is no approach enabling the definition or checking of MT intents. Another uncovered feature is supporting multiple occurrences (i.e., reusing several instances of a MT). This would need mechanisms for composing and synchronizing the multiple MT occurrences, in line with "localized transformations" [25] or "flexible instantiation policies" [26]. More generally, automated mechanisms for composing a MT out of reused partial MTs are not exploited by the analyzed approaches. This is so as all approaches – except patterns – see the reused MT as a black box. In patterns, one can manually compose reused MTs, but none of the approaches have facilities to automate the composition process at the code level. That would require a combination with internal composition techniques like [27,28].

6 Related Work

Reuse of MDE-related artefacts, like metamodels [5] and DSLs [8,22,29], is being actively investigated. In this paper, we have focused on reuse of transformations across metamodels, so-called inter-transformations in [2]. Other kinds of MT reuse include intra-transformation reuse (i.e., reuse within a MT for the same metamodel) and transformation composition. We refer to [2] for further details on these kinds of reuse.

Intra-transformation reuse is typically specific for a transformation language. Some of the proposed techniques include rules with variability [30], ATL module superimposition [31], and rule inheritance [32]. Other internal composition mechanisms are phases, hooks [27] and unit combinators [28]. As mentioned in Sect. 5, an interesting line of work is the combination of inter- and intra-transformation reuse.

Several classifications of MT approaches [33] and tools [34] exist. The features of some MT approaches, like *parameterization* or support for high-order transformations, facilitate reuse. Most reuse approaches are independent of the MT language. However, those that are dependent (like concepts [6]) benefit from the declarative style of the MT language, as it simplifies the rewriting of the MT specification.

For space constraints, we left out a detailed comparison with other reuse approaches across metamodels, like [14–17]. Anyhow, these approaches were taken into account when developing the proposed feature model. Mapping operators [14] are predefined adapters between metamodels, which by themselves define a MT. In [15], a transformation requirement model is extracted from an existing MT to describe the metamodels over which the MT can be reused. This is similar to constraint-based model types [17], but while requirement models use a DSL to express typing requirements, constraint-based model types use logic. Finally, generic MTs [16] are similar to concepts, but specifying relations between the type parameters is not possible, and there is limited support for adaptation [16]. For comparison, we provide the feature model configuration of those approaches at http://bit.ly/bellairs18.

7 Conclusion and Perspectives

To achieve true engineering of MDE solutions, mechanisms to scale up MT to industrial practice – like reuse – are required. In this paper, we have analyzed and classified approaches to *MT reuse across metamodels* in order to clarify the existing reuse options. We have provided a feature model mapping the current option space, and identified gaps that signal opportunities for further research and challenges for the MT community. These include the specification and checking of advanced semantic properties indicating a correct reuse [17], and the combination of intra- and inter-transformation reuse approaches.

In the future, we would like to outline guidelines for selecting the appropriate reuse technique depending on the scenario. We also plan to expand our classification with a goal model to facilitate the decision on the reuse choice, and to open the spectrum to other reuse scenarios. Analyzing how often are MTs reused in practice and detecting reuse opportunities, e.g., using tools like [35], remain as future work.

Acknowledgements. Work partially supported by the Spanish MINECO (TIN2014-52129-R), the R&D programme of the Madrid Region (S2013/ICE-3006), the Safran/Inria/CNRS collaboration GLOSE, and the Inria/CWI Associated Team ALE.

References

1. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2), 131–183 (1992)
2. Kusel, A., et al.: Reuse in model-to-model transformation languages: are we there yet? SoSyM **14**(2), 537–572 (2015)
3. Steel, J., Jézéquel, J.M.: On model typing. SoSyM **6**(4), 401–414 (2007)
4. Guy, C., Combemale, B., Derrien, S., Steel, J.R.H., Jézéquel, J.-M.: On model subtyping. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 400–415. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_30
5. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. SoSyM **12**(3), 453–474 (2013)
6. Cuadrado, J.S., Guerra, E., de Lara, J.: A component model for model transformations. IEEE Trans. Softw. Eng. **40**(11), 1042–1060 (2014)
7. de Lara, J., Guerra, E.: A posteriori typing for model-driven engineering: concepts, analysis, and applications. ACM TOSEM **25**(4), 31:1–31:60 (2017)
8. de Lara, J., Guerra, E., Cuadrado, J.S.: Model-driven engineering with domain-specific meta-modelling languages. SoSyM **14**(1), 429–459 (2015)
9. Ergin, H., Syriani, E., Gray, J.: Design pattern oriented development of model transformations. Comput. Lang. Syst. Struct. **46**, 106–139 (2016)
10. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Programm. **72**(1–2), 31–39 (2008)
11. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69927-9_4
12. Jézéquel, J.-M., Barais, O., Fleurey, F.: Model driven language engineering with kermeta. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 201–221. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18023-1_5
13. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: ICSE, IEEE Computer Society, pp. 485–495 (2009)
14. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schoenboeck, J., Schwinger, W.: Surviving the heterogeneity jungle with composite mapping operators. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 260–275. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13688-7_18
15. de Lara, J., Di Rocco, J., Di Ruscio, D., Guerra, E., Iovino, L., Pierantonio, A., Cuadrado, J.S.: Reusing model transformations through typing requirements models. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 264–282. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_15
16. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 290–304. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30187-5_21
17. Zschaler, S.: Towards constraint-based model types: a generalised formal foundation for model genericity. In: VAO@STAF, pp. 11–18. ACM (2014)

18. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Reverse engineering of model transformations for Reusability. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 186–201. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_14
19. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and Kleisli categories. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 163–177. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28872-2_12
20. Salay, R., Zschaler, S., Chechik, M.: Correct reuse of transformations is hard to guarantee. In: Van Van Gorp, P., Engels, G. (eds.) ICMT 2016. LNCS, vol. 9765, pp. 107–122. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42064-6_8
21. Sun, W., Combemale, B., Derrien, S., France, R.B.: Using model types to support contract-aware model substitutability. In: Van Gorp, P., Ritter, T., Rose, L.M. (eds.) ECMFA 2013. LNCS, vol. 7949, pp. 118–133. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39013-5_9
22. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M.: Melange: a meta-language for modular and reusable development of DSLs. In: SLE, pp. 25–36. ACM (2015)
23. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Trans. Model. Comput. Simul. **12**(4), 290–321 (2002)
24. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G.M., Syriani, E., Wimmer, M.: Model transformation intents and their properties. SoSyM **15**(3), 685–705 (2014)
25. Etien, A., Muller, A., Legrand, T., Paige, R.F.: Localized model transformations for building large-scale transformations. SoSyM **14**(3), 1189–1213 (2015)
26. Morin, B., Klein, J., Kienzle, J., Jézéquel, J.-M.: Flexible model element introduction policies for aspect-oriented modeling. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6395, pp. 63–77. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16129-2_6
27. Sánchez Cuadrado, J., García Molina, J.: Approaches for model transformation reuse: factorization and composition. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 168–182. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69927-9_12
28. Kleppe, A.: MCC: a model transformation environment. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 173–187. Springer, Heidelberg (2006). https://doi.org/10.1007/11787044_14
29. Sutîi, A., van den Brand, M., Verhoeff, T.: Exploration of modularity and reusability of domain-specific languages: an expression DSL in metamod. Comput. Lang. Syst. Struct. **51**, 48–70 (2018)
30. Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., Plöger, J.: Variability-based model transformation: formal foundation and application. Formal Asp. Comput. **30**(1), 133–162 (2018)
31. Wagelaar, D., Straeten, R.V.D., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. SoSyM **9**(3), 285–309 (2010)
32. Wimmer, M., et al.: Surveying rule inheritance in model-to-model transformation languages. JOT **11**(2), 3:1–3:46 (2012)
33. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)

34. Kahani, N., Bagherzadeh, M., R. Cordy, J., Dingel, J., Varro, D.: Survey and classification of model transformation tools. In: SoSyM (2018, in press)
35. Mengerink, J., Serebrenik, A., Schiffelers, R.R.H., van den Brand, M.G.J.: Automated analyses of model-driven artifacts: obtaining insights into industrial application of MDE. In: IWSM-Mensura, pp. 116–121. ACM (2017)

Systematic Recovery of MDE Technology Usage

Juri Di Rocco[1], Davide Di Ruscio[1], Johannes Härtel[2], Ludovico Iovino[3], Ralf Lämmel[2](✉), and Alfonso Pierantonio[1]

[1] Department of Information Engineering, Computer Science and Mathematics, Università degli Studi dell'Aquila, L'Aquila, Italy
{juri.dirocco,davide.diruscio,alfonso.pierantonio}@univaq.it
[2] Faculty of CS, University of Koblenz-Landau, Mainz, Germany
{johanneshaertel,laemmel}@uni-koblenz.de
[3] Gran Sasso Science Institute, L'Aquila, Italy
ludovico.iovino@gssi.it

Abstract. MDE projects may use various MDE technologies (e.g., for model transformation, model comparison, or model/code generation) and thus, contain various MDE artifacts (such as models, metamodels, and model transformations). The details of using the MDE technologies and the relationships between the MDE artifacts are typically not accessible at a higher level of abstraction, which makes it hard to understand, build, and test the MDE projects and thus, to reuse the contained MDE artifacts. In this paper, we present a megamodel-based reverse engineering methodology and an infrastructure MDEprofiler for recovering details of using MDE technologies in MDE projects and modeling these details at a higher level of abstraction. We exemplify the approach for MDE projects that use ATL-based model transformations.

1 Introduction

The Model-Driven Engineering (MDE) community has made considerable progress in recent years with improving productivity and quality in software development. However, cost-efficient adoption of MDE is still a challenge [1]. Introspection about processes and usage of model-driven techniques and technologies may help here, if it enables a higher-level representation of tooling architectures, thereby enhancing understanding and reuse.

Research Problem. MDE projects may use various MDE technologies (e.g., for model transformation, model comparison, or model/code generation) and thus, they contain various artifacts (such as models, metamodels, and model transformations). The details of using MDE technologies and the relationships between the artifacts are typically not accessible at a higher level of abstraction, which makes it hard to understand, build, and test the projects and thus, to reuse the contained artifacts. This problem is arguably very relevant for model repositories [2,3] which, as a result of lacking access to a higher level of abstraction

A. Rensink and J. Sánchez Cuadrado (Eds.): ICMT 2018, LNCS 10888, pp. 110–126, 2018.
https://doi.org/10.1007/978-3-319-93317-7_5

regarding usage of MDE technologies, end up focusing on aggregation of artifacts without attached 'architectural' information.

In principle, one could use a megamodeling approach, up to the point of executable megamodeling scripts [4], for managing MDE projects. The model elements of a megamodel are artifacts such as models, metamodels and transformations. A megamodel also contains relationships between artifacts, for example, conformance and transformation. Thus, megamodeling offers the possibility to specify relationships between artifacts and to navigate between them. For a megamodel to be practically useful though, it would need to address the technological heterogeneity of MDE projects which rely on, for example, mainstream build systems, scripting languages, and test frameworks.

Further, we must not limit ourselves to prescriptive megamodeling or forward engineering; we also need to be able to 'discover' megamodels and 'recover' their instances systematically, semi-automatically, and efficiently so that we can benefit from them without much extra developer effort. Thus, we face a problem similar to *software architecture reverse engineering* or *architecture recovery* [5,6] in that software projects may lack higher-level architectural descriptions. Recovery is to be leveraged when a suitable description has never existed or it is no longer 'in sync' with the actual code. In an MDE technological context, we may be interested in architectural knowledge such as model artifacts in a project, more specific types of models (e.g., metamodels), model-to-metamodel conformance, applications of model-management operations (e.g., model transformation, model/code generation, model merging, model weaving, model comparison, and model patching), evolution-related relationships, and some types of technological traces, for example, build scripts, launcher configurations, or tests.

Previous Work by (Some of) These Authors. The software language repository YAS manages a technologically heterogeneous project by a suitable megamodel, as described in [7], but this work does not address 'mainstream MDE', neither does it address reverse engineering. Megamodeling is discussed for MDE technologies (including EMF, ATL, and Xtext) in [8], but reverse engineering is not addressed, despite being stated as a direction for future work. A rule-based approach to mining artifact relationships with an application to EMF is presented in [9], but no methodology for discovering megamodels is provided. All of this previous work invokes the term 'linguistic architecture' [10] as a form of megamodeling and a form of software architecture.

Contributions of the Paper. We present a megamodel-based reverse engineering methodology and an infrastructure MDEprofiler for recovering details of using MDE technologies in MDE projects. We exemplify the approach for MDE projects that exercise ATL-based model transformations. Our experimental validation is limited to ATL and 'implied' technologies or languages such as Ecore, KM3, Ant, and launcher configurations, but our approach is completely 'generic'. Our methodology is designed to support the iterative process of discovering a pool of recovery heuristics for megamodel elements. We demonstrate the methodology with the ATL Zoo.[1]

[1] https://www.eclipse.org/atl/atlTransformations/.

Road-Map of the Paper. Section 2 describes our methodology for recovering MDE-technology usage. Section 3 describes our infrastructure for recovery. Section 4 evaluates our approach by means of a case study for the ATL Zoo. Section 5 discusses related work. Section 6 concludes the paper.

2 Recovery Methodology

Figure 1 summarizes key aspects of our methodology. Any number of MDE projects (possibly also adding new ones over time) are analyzed semi-automatically to recover megamodels representing MDE-usage information. Heuristics are used to locate artifacts of interests (e.g., models) and artifacts that encode relationships (e.g., build scripts with model transformation applications). The recovered megamodels are essentially graphs with artifacts of interest as nodes and relationships as edges. Simple measures are computed for the megamodels. In particular, 'dangling' nodes are determined, as they are considered indicators of missing relationships. Domain knowledge and technology documentation are leveraged to manually refine the applied heuristics and to conceive new ones until all the artifacts of the analyzed MDE projects are modeled together with the corresponding relationships. This recovery process is intrinsically incremental. In the sequel, we discuss artifacts in MDE projects, relationships between them, and heuristics for relationship inference in more detail.

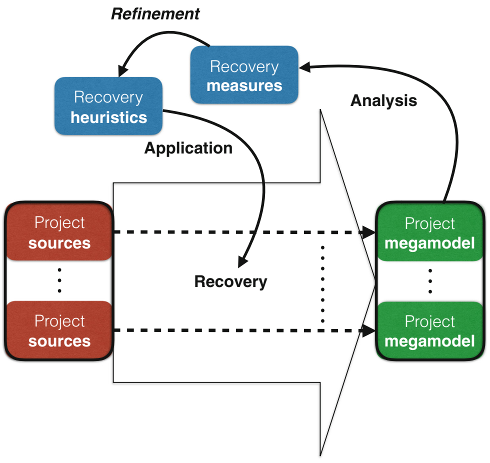


Fig. 1. Megamodel-based reverse engineering.

2.1 Artifacts in a MDE Project

As shown in Fig. 2 and described below, several kinds of artifacts are considered when applying MDE. *Available artifacts* make up the system in terms of its source code and other resources that are available typically through version control or download. *All artifacts* includes artifacts that may be *not at all or not directly available*. For instance, an artifact may only be obtainable by system building or testing. Also, an artifact may only be transient (e.g., as a runtime object during the execution of a testcase). Further, an artifact may only be obtainable by some well-defined computational step, for instance the application of a code generator — with or without this application being exercised by build management or testing. Yet further, an artifact may be unavailable, but its existence, at least, in the past, is known simply because there are traces of it (i.e., references to it) in the available artifacts. *Artifacts of interest* are those (available or not) that are obviously of interest for recovering technology usage. In the case of ATL-based model transformation, artifacts of interest are clearly the ATL transformations themselves, but also source and target models for transformations as well as metamodels for conformance. *Artifacts with traces* are those (available) artifacts (of interest or not) in which we may locate traces to artifacts (mainly references). Subject to a classification of the artifacts with traces, these artifacts may be interpreted as (encoding) relationships between artifacts.

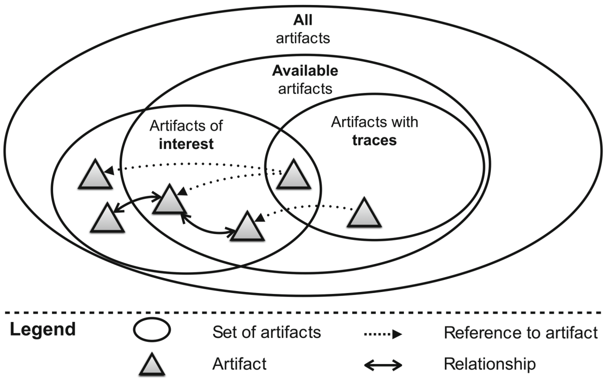


Fig. 2. Artifacts in a MDE project.

The overall assumption is that we may identify artifacts of interest by examining algorithmically the available artifacts and we may identify relationships between artifacts by examining, again, algorithmically available artifacts on the grounds of technology-specific patterns for traces.

2.2 Relationships to be Recovered

Figure 3 identifies 'abstract' artifacts of interest with relationships for the running example of ATL. In particular, there are source and target models, the corresponding metamodels (MMs), the actual ATL model transformation (MT), and the application thereof. We also show relationships between these artifacts that need to be recovered. Relationships between artifacts, e.g., conformance and transformation application in the example, can be identified in different ways:

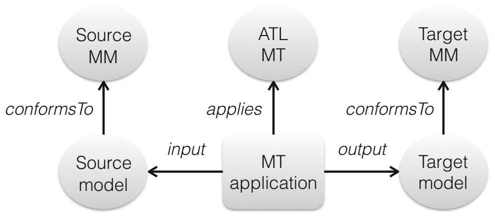


Fig. 3. 'Abstract' artifacts and relationships for ATL usage.

Trace-Based Identification. Based on the type of referring artifact (e.g., an ANT file), based also on the details of reference (e.g., the argument position of an ATL transformation execution), one may identify a relationship (e.g., a model to serve as the 'source' of a model transformation).

Computational Identification. By considering a more or less standardized, technology-specific functionality (e.g., the operation for Ecore-based conformance checking) on given candidate artifacts (e.g., a model artifact and a metamodel artifact), one may identify a relationship (e.g., conformance).

Mining-Based Identification. Based on a more 'ad-hoc' application of technology-specific functionality (e.g., a comparison of vocabulary extracted from various artifacts) on given candidate artifacts, one may identify a relationship (e.g., similarity).

2.3 Heuristics for Recovery

Heuristics for identifying artifacts of interest and finding traces for relationships are based on the following techniques mostly inspired by existing work on reverse engineering and megamodeling.

Filename Heuristics. Many types of artifacts may be precisely detected on the grounds of filenames or extensions thereof [11]. For instance, the '.atl' extension identifies an ATL model transformation — especially within an MDE project. Clearly, filenames may not always be sufficient; one may also need to consult the content of files for the purpose of artifact classification. For instance, EMF models may be stored in '.xmi' files, but other extensions are also used.

Watermark Heuristics. Some types of artifacts may be precisely detected by looking for specific content patterns ('watermarks') in files [11,12]. For instance, a syntax definition for the EMFText technology would be a '.cs' file that contains the string 'syntaxdef' [12]. (The extension '.cs' alone would be imprecise, if we assume that C# files could also be in the same project.)

Parser Heuristics. Some types of artifacts may be precisely detected by just trying to parse the artifact by a standard component for the type of interest. For instance, an XML file could be precisely detected, by just invoking any XML parser, e.g., a DOM-based one, on the file in a non-lax mode. A filename or watermark heuristic can be used as a precondition, if costs of parsing are a concern [13].

Component Heuristics. Some types of artifacts may be precisely detected and some types of suspected relationships may be precisely verified by reusing the technology of interest, or rather a component thereof [8,13]. For instance, a suspected conformance relationship may be verified by the available component (operation) for Ecore-based conformance checking, as discussed in Sect. 2.2.

Extractor Heuristics. Customized fact extractors [13–15] may be used to identify traces in given artifacts, thereby helping with recovery of relationships. For instance, a heuristics for ANT files may extract instances of common patterns of using ANT for applying model transformations.

Analyser Heuristics. Ultimately, more advanced software analyses may be used to detect or verify relationships. For instance, one may infer source and target metamodels (or approximations thereof) from model transformations [16], thereby preparing the detection of potential source or target models on the grounds of attempted conformance checking.

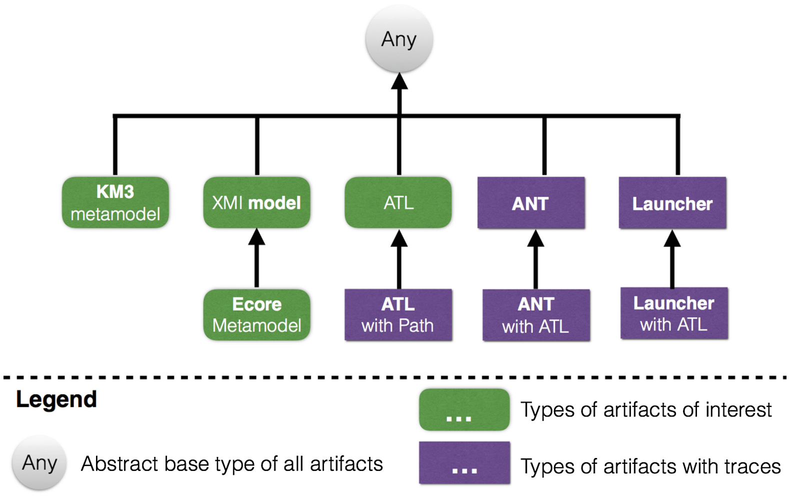


Fig. 4. Artifacts involved in the recovery of ATL usage. (Color figure online)

Figure 4 arranges some of the heuristics that were developed in the case study of Sect. 4. The root node is 'abstract'; it does not correspond to any actual heuristic. The rounded (green) shapes correspond to heuristics for detecting available artifacts of interests. The angular (purple) shapes correspond to heuristics for artifacts with potential traces. The heuristics are arranged in a specialization hierarchy to express that a sub-heuristic should only be tried once the super-heuristic was confirmed. For instance, we first try to find all models and then we filter out all metamodels among them.

The key principle of the methodology is that heuristics like those in Fig. 4 are introduced in an iterative process on the grounds of measuring connectiveness of the recovered graph and leveraging domain knowledge (regarding MDE technologies) for identifying opportunities for relationship recovery by additional heuristics.

3 The Recovery Infrastructure

In this section, the recovery infrastructure supporting the methodology presented in the previous section is described. As shown in Fig. 5, the implemented recovery machinery consists of three main components, namely `RepositoryConnector`, `HeuristicsManager`, and `MegamodelVisualizer`.

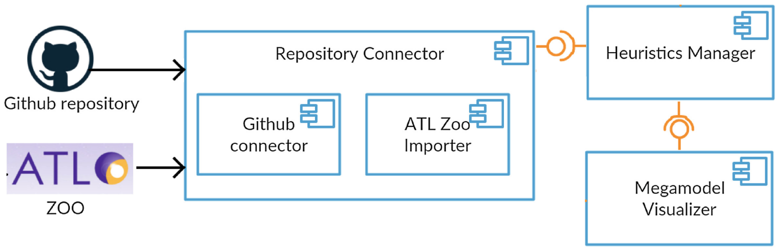


Fig. 5. Components of the recovery architecture.

The `RepositoryConnector` connects to data sources that export reusable MDE projects, which thus can be locally downloaded for subsequent analysis. Currently, the recovery infrastructure can import data from the ATL Zoo and from GitHub repositories (see Sect. 3.1). `RepositoryConnector` is extensible in that it provides developers with interfaces that can be implemented for adding new connectors.

The `HeuristicsManager` component is in charge of applying the available heuristics on all the projects locally downloaded by `RepositoryConnector`. The outcome of the recovery process consists of models conforming to a specifically conceived metamodel as presented in Sect. 3.2. The outcome of `HeuristicsManager` can be consumed in different ways — including the possibility of graphically visualizing it in order to give an overview of the analyzed projects and to support the understanding of the contained artifacts. The `MegamodelVisualizer` component presented in Sect. 3.3 takes recovery models as input and generates a graphical representation of them.

3.1 Repository Connector

In order to enable the analysis of MDE projects, our infrastructure downloads the projects. Currently, the infrastructure can import projects from the ATL Zoo and from GitHub repositories. The ATL Zoo is a widely used repository of model transformations, which have been the subject of several empirical works over the last few years. Unfortunately, the repository does not provide a dedicated API to easily export the available projects. Thus, HTML scraping is the only viable way to programmatically download the data available in the repository. The GitHub connector exploits the Git API[2] for locally cloning a project of interest identified by its `owner` and `name` attributes.

[2] https://developer.github.com/v3/.

3.2 Heuristics Manager

Once data has been downloaded by means of the available connectors, the actual recovery process starts. The outcome of the process is a model conforming to a specifically conceived *recovery metamodel*. The heuristics currently available are presented later in this section.

The Recovery Metamodel. As mentioned earlier, the model generated by the recovery process is a *graph* consisting of *nodes* and *edges*. For each artifact that can be identified by the available heuristics, the recovery approach generates a corresponding target node. The recovery process also detects relationships among artifacts. Detected relationships are represented as edges among previously recovered nodes. For instance, a model transformation consuming models conforming to a source metamodel and generating models conforming to a target metamodel give rise to a sub-graph consisting of nodes and edges as follows. One node would represent the analyzed transformation. Two edges would link the transformation with two further nodes representing the source and target metamodels.

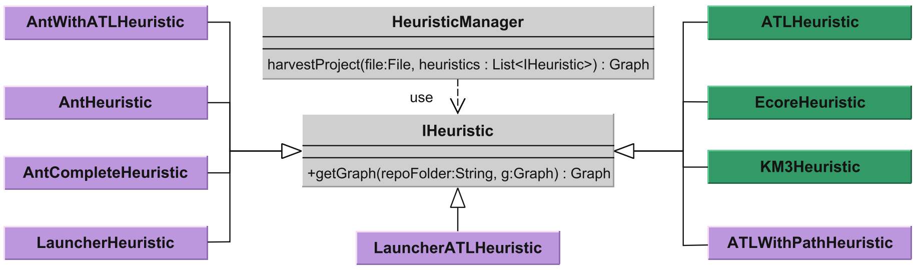


Fig. 6. Class diagram showing an overview of the Heuristic Manager. (Color figure online)

Recovery Heuristics. Figure 6 shows a class diagram representing the hierarchical organization of the heuristics currently available in the `HeuristicsManager` component shown in Fig. 5. Each heuristic implements the `Heuristic` interface or extends an available implementation. In Fig. 6, the elements `ATLHeuristic`, `EcoreHeuristic`, and `KM3Heuristic` are in green color in order to be consistent with what is discussed in the previous section. These heuristics identify artifacts of interest, i.e., ATL transformations and metamodels specified either in KM3 or Ecore. Heuristics that are shown in Fig. 6 with the violet color represents heuristics that have been implemented in order to recover relationships among transformations, models, and metamodels.

Listing 1. Fragment of ATLHeuristic

```
1  package it.univaq.MDEProfiler.heuristic;
2  ...
3  public class ATLHeuristic implements IHeuristic {
4    private String extension = ".atl";
5    private String nodeKind = "NodeType.ATL";
6    @Override
7    public Graph getGraph(String repoFolder, Graph g){
8          File repoFolderF = new File(repoFolder);
9          List<File> fList = FileUtils.getFilesByEndingValue(repoFolderF,
10         extension);
11         for (File file : fList) {
12             boolean guard = g.getNodes().stream()
13                 .anyMatch(s -> s.getUri().equals(file.getAbsolutePath()));
14             if(!guard) {
15                 Node n = GraphFactory.eINSTANCE.createNode();
16                 n.setDerivedOrNotExists(false);
17                 n.getType().add(nodeKind);
18                 n.setUri(file.getAbsolutePath());
19                 n.setName(file.getName());
20                 g.getNodes().add(n);
21             }
22         }
23         return g;
24     }
25 }
```

Listing 1 shows a fragment of the Java implementation of `ATLHeuristic`. Essentially, in each project, the heuristic searches for files having the .atl extension (see line 4), and for each of them a new node typed `NodeType.ATL` is generated in the target recovery model (see line 5 and lines 13–20). Similarly, `KM3Heuristic` and `EcoreHeuristic` search for .km3 and .ecore files, respectively and generate target `NodeType.KM3` and `NodeType.Ecore` nodes accordingly.

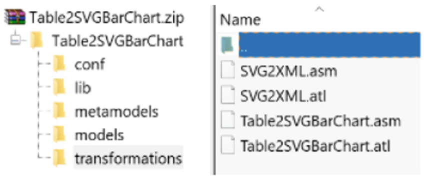
The recovery of relationships among generated nodes requires more elaborated analyses that should consider additional artifacts like ANT scripts and launcher files. For instance, Listing 2 shows the launch file configuration for the ATL transformation *Families2Persons*[3]. Lines 6–15 contain precious information about the input and target elements of the ATL *Families2Persons* transformation, which if considered alone does not contain such details.

Listing 2. A sample ATL launch file

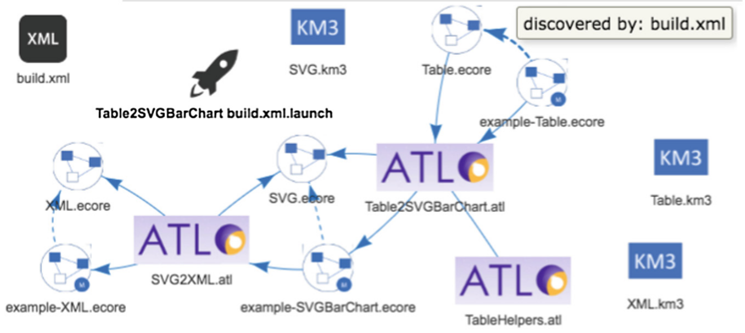
```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <launchConfiguration ..>
3      <stringAttribute key="ATL File Name"
4      value="/Families2Persons/Families2Persons.atl"/>
5      ...
6      <mapAttribute key="Path">
7          <mapEntry key="Families"
8              value="/Families2Persons/Families.ecore"/>
9          <mapEntry key="IN"
10             value="/Families2Persons/sample-Families.xmi"/>
11         <mapEntry key="OUT"
12             value="/Families2Persons/sample-Persons.xmi"/>
13         <mapEntry key="Persons"
14             value="/Families2Persons/Persons.ecore"/>
15     </mapAttribute>
16 </launchConfiguration>
```

[3] https://www.eclipse.org/atl/atlTransformations/#Families2Persons.

The analysis of ATL launch configuration files like the one shown in Listing 2 is implemented by the `LauncherATLHeuristic` shown in Fig. 6. Due to space limitation, this paper does not give more details about the implementation of the currently available heuristics. Interested reader can refer to the Github project related to this work[4].



(a) Content of the project as it is available from the ATL Zoo



(b) Graphically representation of the recovered model

Fig. 7. The Table 2SVGBarChart project.

3.3 Megamodel Visualizer

The recovered model generated for each project can be processed by other services, for example, to graphically represent the automatically recovered project as shown in Fig. 7b. By looking at such a model, users can get a clear understanding about how the different elements are connected. By contrast, Fig. 7a shows the folders contained in the package of the *Table 2SVGBarChart* project[5], as users could explore the project by means of a file explorer and view the content of files to understand how different artifacts are related. We contend that the visualized megamodel helps much better with understanding.

The *MegamodelVisualizer* component shown in Fig. 5 is in charge of generating diagrams like the one shown in Fig. 7b by means of an Acceleo[6]-based generator; it takes a recovery model as input and generates HTML5+Javascript code. The generated code uses the Visjs[7] Javascript library and it can han-

[4] https://github.com/MDEGroup/MDEProfile.
[5] https://www.eclipse.org/atl/atlTransformations/#Table2SVGBarChart.
[6] https://www.eclipse.org/acceleo/.
[7] http://visjs.org.

dle large amounts of dynamic data while enabling manipulation, representation, and interaction. For instance, in the diagram shown in Fig. 7b, the artifact *XML.ecore* is visually associated with the Ecore type and the link with the artifact *example-XML.ecore* highlights that the latter is a model conforming to the former. Moreover, the ATL transformation *SVG2XML.atl* takes as input the *XML.ecore* node as metamodel and the *example-XML.ecore* element as model. The output consists of the *example-SVGBarChart.ecore* model conforming to the *SVG.ecore* metamodel. The node *build.xml* contributes the discovery of the represented relationship as shown by the hovering label *discovered by*.

4 Case Study

This section discusses the application of the proposed approach to the ATL Zoo consisting of ≈100 model transformation projects. The case study was performed in an iterative process in order to gradually add heuristics for new types of nodes and edges. Initially, we implemented some heuristics to identify 'obvious' artifact types of interest. Subsequently, we went through some iterations to add heuristics to recover relationships among previously discovered nodes. The case study addresses the following research questions:

– **RQ1:** What is the accuracy of recovered models?
– **RQ2:** How much effort is saved by automated recovery?

Evaluation Measures. We use *precision* and *recall* measures as follows:

$$precision = \frac{Corr_a}{All_a} \tag{1}$$

$$recall = \frac{Corr_a}{All_m} \tag{2}$$

where $Corr_a$ is the *correct number* of elements recovered by the approach, All_a is the total number of elements *automatically* produced by the approach, and All_m is the expected total number of elements as produced by a *manual* harvesting phase.

Results. Table 1 shows representative results related to each iteration of the performed case study. In the first five iterations, we gradually added heuristics to discover Ecore, ATL, KM3, and ANT files. All the artifacts of interest were dangling (see the `#Edges` and `#DanglingNodes` columns). This means that we were able to increasingly discover new types of elements even though they were added in the recovery model as nodes without edges. The addition of heuristics for analyzing ATL launcher file configurations and ANT scripts for ATL automation led to a turning point. That is, even though new nodes were discovered, the number of dangling ones was decreased.

Table 1. Case-study results

Iteration	Applied heuristics	#Nodes	#Edges	#Dangling nodes
1	EH	327	0	327
2	EH, AH	587	0	587
3	EH, AH, KH	795	0	795
4	EH, AH, KH, LH	887	0	887
5	EH, AH, KH, LH, ANH	1001	0	1001
6	EH, AH, KH, LH, ANH, APH	1001	37	965
7	EH, AH, KH, LH, ANH, APH, LTH	1041	236	887
8	EH, AH, KH, LH, ANH, APH, LTH, ANATLH	1133	533	745

Legend: EH: EcoreHeuristic, AH: ATLHeuristic, KH: KM3Heuristic, LH: LauncherHeuristic ANH: ANTHeuristic, APH: ATLWithPathHeuristic, LTH: LauncherATLHeuristic ANATLH: ANTWithATLHeuristic

Figure 8 graphically represents the effect of applying the heuristics by focusing on the discovered and dangling nodes. The chart shows how considering specific files and properties leads to the discovery of new relationships. Starting at iteration 6, new nodes were discovered with a consequent reduction of dangling ones.

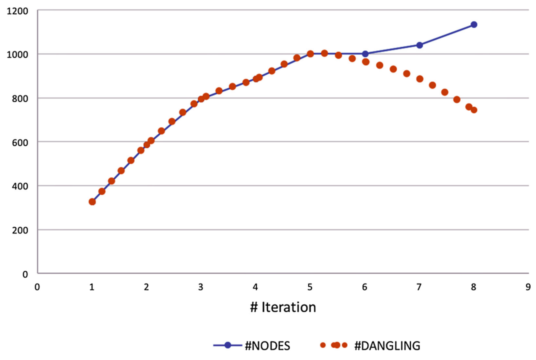


Fig. 8. Nodes recovered during the case study.

Reducing the number of dangling nodes is a challenging task, which requires the implementation of new heuristics able to cover new node types and relationships by deducing additional information from the available artifacts. For instance, by looking at the dangling nodes at the end of the last iteration shown in Table 1 we noticed that many of them are of KM3 type and many ATL transformations are defined on Ecore metamodels that can be automatically generated from the available KM3 specifications. For instance, in the case shown in Fig. 7b there are three dangling nodes of type KM3 (i.e., *XML.km3*, *Table.km3*, and *SVG.km3*). By applying the ATL transformation *KM32Ecore* [17] on such KM3 specifications, we noticed that the obtained Ecore metamodels are those already available in the project (i.e., XML.ecore, Table.ecore, and SVG.ecore). By applying such a heuristic, the three KM3 nodes shown in Fig. 7b would be removed from the list of dangling nodes. The ATL Zoo contains 73 projects that have similar cases. If all of them are managed, as previously discussed, the number of dangling nodes would decrease from 745 to 347.

To evaluate the accuracy of the approach and thus, to answer *RQ1*, we manually analysed 40 projects (randomly) downloaded from the ATL Zoo. In particular, a senior modeler manually inspected such projects (without knowing in advance the results of the tools) and recovered the nodes and relations of the corresponding megamodels[8].

The case study's accuracy can be increased by means of adding heuristics. For instance, the analysed projects contain TCS specifications [18], which are currently not covered by MDEprofiler and this is reflected by the *precision* and *recall* measures.

To answer *RQ2*, the 40 projects considered to produce the data in Table 2 have been analysed by means of MDEprofiler executed on an Intel Core i5 machine with 8 GB of RAM. The analysis took ≈10 s, whereas the senior modeler needed 2 full-time working days to perform the analysis on the same data set.

Table 2. Precision and recall of recovery.

	Nodes	Relations
Precision	0.913	0.896
Recall	0.942	0.636

5 Related Work

We begin with a discussion of heuristics used for recovery purposes in the domains of (a) architecture recovery, (b) traceability recovery, and (c) analysis of software, technology or language usage. Afterwards, we discuss related work on megamodels in model-management systems.

Heuristics for Architecture Recovery. Bowman et al. [19] compare three recovered architectures: a conceptual architecture based on the documentation, a concrete architecture that is derived from the actual system, and an ownership architecture extracted from version control. By examining the overlap of edges, they check whether one architecture correlates with another. Concrete, ownership and conceptual architecture recovery can be considered as a kind of heuristic. In contrast, our work combines the output of heuristics and refines the set of used heuristics through an iterative process. While Bowman et al. considers fundamentally different sources, in [20] a very fine-grained and specific set of heuristics on code-package structures is employed to guide exploration of system architecture. Our work also facilitates fine-grained exploration, by means of an extensible heuristics-based mechanism. In [21], source code is represented as a graph of, e.g., variables, types, or import relations. Here, heuristics are used in the form of patterns that are matched on this graph. These patterns contain placeholders for abstract components and connectors. An approximate instantiation on the source graph produces the resulting architecture. The methodology comes close to ours in that it facilitates domain knowledge in an iterative and

[8] A replication package consisting of the MDEprofiler tool, the analysed projects, and of the obtained results is available for download at https://github.com/MDEGroup/MDEProfile.

interactive process to define the patterns. Our approach recovers megamodels of actual systems based on file-type recognition. This motivates our need for flexible heuristics that we implement in plain Java. In [22], the authors compare a set of alternatives to group the system using hierarchical clustering and conclude on their characteristics (e.g., one way of clustering is good for detecting utility functions). Depending on which similarity definition is chosen for clustering, this method can be seen as very general and domain-independent heuristics for grouping and connecting nodes. Architecture recovery of web applications facilitated by different extractors is pursued in [23] with a form of extractors comparable to our heuristics. The extractors also provide relationships for the web application.

Heuristics for Traceability Recovery. Traceability recovery concentrates on mining edges between artifacts. Here, the usage of language-agnostic heuristics is very common, since trace links often reside between artifacts in different languages including natural language. For instance, in [24], links are recovered by computing the cosine similarity between the artifact term vectors. The recovered trace links connect Java and functional requirements as well as C++ and manual pages. Alternatively, in [25], sequential pattern mining is applied on commits to connect any type of artifact in a repository co-occurring in a change. We see such types of generic heuristics as a promising extension to our approach, especially to uncover unknown domain-specific heuristics. In this paper we concentrate on ATL-specific recovery.

Heuristics for Software, Technology, and Language Usage. In [12], the usage of Eclipse-based MDE technologies in projects hosted on GitHub is analyzed by counting the files that are strongly related to technology usage. Another language-usage analysis of repositories, without being focused on MDE, is described in [26]. The authors also use file extensions as a heuristic to detect languages. We use file extensions only as the simplest heuristic. API usage in projects, as a very specific kind of software usage, is analyzed extensively in related work (e.g., [27–29]). Different features or metrics are used for characterizing API usage, for example, whether or not a component uses a given API or whether or not the component extends or simply reuses the API.

Megamodeling and Executable Model Management. Megamodels, as introduced in [30], are concerned with models as first-class entities. Megamodels are often used in executable model management systems to organize tasks on models, e.g., the application of transformations, querying, merging, and constraint checking. For instance, in [31], an explorative framework for working with models is described that follows the megamodeling principles. Alternatively, in [4], a layer on top of heterogeneous repositories is presented to get uniform model-based access to the system by writing model operations in a DSL. In [32], graphical and interactive support is described; this work is close to our model visualizer. There is no related work on megamodels where heuristics are used for identification of model elements and recovery of relationships. In some of our previous work on megamodeling [8,9,13], we considered heuristics, but without a methodology for their discovery along an iterative process.

6 Conclusion and Future Work

MDE projects are typically shared without any machine-readable description. Projects are given as packages consisting of files, possibly organized in folders, that modelers have to manually scan in order to figure out how the different project artifacts are related. Thus, understanding the artifacts contained in MDE projects and their relationships can be a strenuous and error-prone activity.

In this paper, we presented an approach based on megamodels which permits to automatically recover the structure of MDE projects represented as typed nodes and relationships among them. The approach is implemented as the recovery infrastructure MDEprofiler. The approach has been applied in a case study on the widely used ATL Zoo consisting of ≈100 model transformation projects. In future work, we plan to apply the approach to MDE projects retrieved from elsewhere, e.g., GitHub, and to implement additional heuristics, as needed in order to minimize the number of dangling nodes and improve the overall accuracy of the approach. We are also working on extending the portfolio of MDE technologies beyond the current focus on ATL.

References

1. Tomassetti, F., Torchiano, M., Tiso, A., Ricca, F., Reggio, G.: Maturity of software modelling and model driven engineering: a survey in the Italian industry. In: Proceedings of the EASE, pp. 91–100 (2012)
2. Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Model repositories: will they become reality? In: Proceedings of the CloudMDE@MoDELS 2015. CEUR Workshop Proceedings, vol. 1563, pp. 37–42 (2016)
3. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Collaborative repositories in model-driven engineering. IEEE Softw. **32**, 28–34 (2015)
4. Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., Cabot, J.: MoScript: a DSL for querying and manipulating model repositories. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 180–200. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28830-2_10
5. Stringfellow, C., Amory, C.D., Potnuri, D., Andrews, A.A., Georg, M.: Comparison of software architecture reverse engineering methods. Inf. Softw. Technol. **48**, 484–497 (2006)
6. Krikhaar, R.L.: Reverse architecting approach for complex systems. In: Proceedings of the ICSM, pp. 4–11. IEEE (1997)
7. Lämmel, R.: Relationship maintenance in software language repositories. Art Sci. Eng. Program. J. **1**, 27 (2017)
8. Härtel, J., Härtel, L., Heinz, M., Lämmel, R., Varanovich, A.: Interconnected linguistic architecture. Art Sci. Eng. Program. J. **1**, 27 (2017)
9. Härtel, J., Heinz, M., Lämmel, R.: EMF patterns of usage on GitHub. In: Proceedings of the ECMFA. LNCS. Springer (2018, to appear)

10. Favre, J.-M., Lämmel, R., Varanovich, A.: Modeling the linguistic architecture of software products. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 151–167. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_11
11. Favre, J., Lämmel, R., Leinberger, M., Schmorleiz, T., Varanovich, A.: Linking documentation and source code in a software chrestomathy. In: Proceedings of the WCRE, pp. 335–344. IEEE (2012)
12. Kolovos, D.S., Matragkas, N.D., Korkontzelos, I., Ananiadou, S., Paige, R.F.: Assessing the use of eclipse MDE technologies in open-source software projects. In: Proceedings of the OSS4MDEMODELS. CEUR Workshop Proceedings, vol. 1541, pp. 20–29 (2015)
13. Lämmel, R., Varanovich, A.: Interpretation of linguistic architecture. In: Cabot, J., Rubin, J. (eds.) ECMFA 2014. LNCS, vol. 8569, pp. 67–82. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09195-2_5
14. Murphy, G.C., Notkin, D.: Lightweight lexical source model extraction. ACM Trans. Softw. Eng. Methodol. **5**, 262–292 (1996)
15. Ferenc, R., Siket, I., Gyimóthy, T.: Extracting facts from open source software. In: Proceedings of the ICSM, pp. 60–69. IEEE (2004)
16. de Lara, J., Di Rocco, J., Di Ruscio, D., Guerra, E., Iovino, L., Pierantonio, A., Cuadrado, J.S.: Reusing model transformations through typing requirements models. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 264–282. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_15
17. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: Companion to the 21st ACM SIGPLAN OOPSLA 2006, pp. 602–616. ACM (2006)
18. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the GPCE, pp. 249–254. ACM (2006)
19. Bowman, I.T., Holt, R.C.: Software architecture recovery using Conway's law. In: Proceedings of the CASCON, p. 6. IBM (1998)
20. Lungu, M., Lanza, M., Gîrba, T.: Package patterns for visual architecture recovery. In: Proceedings of the CSMR, pp. 185–196. IEEE (2006)
21. Sartipi, K., Kontogiannis, K.: On Modeling software architecture recovery as graph matching. In: Proceedings of the ICSM, pp. 224–234. IEEE (2003)
22. Maqbool, O., Babri, H.A.: Hierarchical clustering for software architecture recovery. IEEE Trans. Softw. Eng. **33**, 759–780 (2007)
23. Hassan, A.E., Holt, R.C.: Architecture recovery of web applications. In: Proceedings of the ICSE, pp. 349–359. ACM (2002)
24. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D.: Information retrieval models for recovering traceability links between code and documentation. In: ICSM, pp. 40–49. IEEE (2000)
25. Kagdi, H.H., Maletic, J.I., Sharif, B.: Mining software repositories for traceability links. In: ICPC, pp. 145–154. IEEE (2007)
26. Karus, S., Gall, H.C.: A study of language usage evolution in open source software. In: Proceedings of the MSR, pp. 13–22. ACM (2011)
27. Lämmel, R., Pek, E., Starek, J.: Large-scale, AST-based API-usage analysis of open-source Java projects. In: SAC, pp. 1317–1324. ACM (2011)
28. Lämmel, R., Linke, R., Pek, E., Varanovich, A.: A framework profile of .NET. In: Proceedings of the WCRE, pp. 141–150. IEEE (2011)
29. Roover, C.D., Lämmel, R., Pek, E.: Multi-dimensional exploration of API usage. In: Proceedings of the ICPC, pp. 152–161. IEEE (2013)

30. Bézivin, J., Jouault, F., Valduriez, P.: On the need for Megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development Workshop (2004)
31. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003 and MDAFA 2004. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005). https://doi.org/10.1007/11538097_3
32. Sandro, A.D., Salay, R., Famelis, M., Kokaly, S., Chechik, M.: MMINT: a graphical tool for interactive model management. In: Proceedings of the MoDELS 2015 Demo and Poster Session. CEUR Workshop Proceedings, vol. 1554, pp. 16–19 (2016)

Technical Debt in Model Transformation Specifications

Kevin Lano[1](✉), Shekoufeh Kolahdouz-Rahimi[2], Mohammadreza Sharbaf[2], and Hessa Alfraihi[1]

[1] Department of Informatics, King's College London, London, UK
{kevin.lano,hessa.alfraihi}@kcl.ac.uk

[2] Department of Software Engineering, University of Isfahan, Isfahan, Iran
{sh.rahimi,m.sharbaf}@eng.ui.ac.ir

Abstract. Model transformations (MT), as with any other software artifact, may contain quality flaws. Even if a transformation is functionally correct, such flaws will impair maintenance activities such as enhancement and porting. The concept of *technical debt* (TD) models the impact of such flaws as a burden carried by the software which must either be settled in a 'lump sum' to eradicate the flaw, or paid in the ongoing additional costs of maintaining the software with the flaw. In this paper we investigate the characteristics of technical debt in model transformations, analysing a range of MT cases in different MT languages, and using measures of quality flaws or 'bad smells' for MT, adapted from code measures.

Based on these measures we identify significant differences in the level and kinds of technical debt in different MT languages, and we propose ways in which TD can be reduced.

1 Introduction

This paper will investigate the issue of *technical debt* (TD) [14] in model transformations (MT). Technical debt refers to the short and long-term impact of software quality flaws such as duplicated code. The *principal* cost of TD is incurred when refactoring or other redesign is used to remove the TD from the software, whilst the *interest* is paid in the additional cost due to the TD each time the software is maintained.

The concept of TD was initially applied to code artifacts, but can also be extended to analysis and design models [3].

In the MDE context, model transformations are a key software resource, which enable MDE processes such as the production of software and documentation from models, the synchronisation of models, and model comparison. Thus the quality and maintainability of MT are likely to be important factors in the successful use of MDE.

The high-level goal of our research is to *quantify and characterise the nature of technical debt in model transformations*. We will adopt the goal-question-metric (GQM) approach of [4] to decompose this goal into specific questions and metrics. The goal leads to the following research questions:

RQ1: What is the prevalence (flaw density) of TD in MT cases?
RQ2: What are the most frequent forms of quality flaw in MT cases?
RQ3: Does the level and character of TD vary between MT languages and between MT categories?
RQ4: Is there a difference between TD prevalence in MT languages and in traditional programming languages?

The questions imply that a significant sample of transformations must be surveyed, for a range of transformation languages and categories. We will use published and machine-readable transformation cases, and public repositories of transformations. Only cases where the complete code of the transformations is available will be considered. We survey the ATL and QVT-R transformation languages because these are the most widely-used MT languages by practitioners [5]. We also consider ETL and UML-RSDS, which are MT languages with distinctive features (implicit invocation in ETL; no rule-rule dependencies in UML-RSDS) whose impact on TD levels is of interest.

2 Metrics for Technical Debt

Following on from the research questions, we need to find concrete measures which quantify the aspects (TD and categories of TD) which the questions refer to. Measures of various ‘bad smells’ or quality flaws are typically used as metrics of TD in code. However, these need adaptation when used for declarative or hybrid MT specification languages: MT specifications define their effect in a less procedural manner than code, they are usually more concise, and are structured based upon rules and operations instead of upon classes and objects. Therefore we define measures specific to MT specifications, adapting TD measures *Excessive Class Length*, *Excessive Method Length*, *Excessive Number of Parameters*, *Duplicate Code*, *Cyclomatic Complexity*, *Coupling Between Objects*, *Too Many Methods* to the MT context.

Based on our experience of developing and maintaining MT specifications, we considered that the following were the most significant factors in impeding the understanding and maintenance of MT specifications: size; semantic complexity (of expressions, rules and operations); complexity of relationships and dependencies between rules/operations; redundancy. These impact the Analysability, Changeability and Testability quality characteristics of software as defined in the ISO/IEC 25010 quality model [8]. In practice they manifest as:

– Excessively large transformations, with many rules/operations and/or high total length (MT size factor). Measured by *ETS*, *ENR*, *ENO*, defined below.
– Unclear rule precedence or execution order (MT rule dependency factor). Measured by *UEX*.
– Excessively complex expressions (MT semantic complexity factor): *ETS*, *ERS*, *EHS*.
– Excessive rule or operation length (MT size factor): *ERS*, *EHS*.

– Excessive numbers of parameters/auxiliary variables for a rule, transformation or operation (MT semantic complexity factor): *EPL*.
– Duplicated expressions or code (MT redundancy factor): *DC*.
– Complex rule or code logic (MT semantic complexity factor): *CC*.
– Complex calling relations between rules, especially cyclic relations (self or mutual recursion). Inheritance of rules/operations is also counted as a dependency of the generalised rule/operation upon the specialised rules/operations (MT rule dependency factor): *CBR*.
– Excessive numbers of rules/operations called from one rule or operation (MT rule dependency factor): *EFO*.

The size of software artifacts is often measured in terms of lines of code (LOC). We prefer to adopt a measure $c(\tau)$ of the semantic content of a model transformation specification τ, based on the complexity of expressions/activities in the transformation. Unlike LOC, this is independent of code formatting style or white space. Each of ATL, ETL, QVT-R and UML-RSDS have similar expression languages based on OCL, and ATL, ETL and UML-RSDS have similar activity languages. Therefore $c(\tau)$ can be defined consistently for all these languages. Table 1 summarises the semantic complexity measure $c(e)$ for some OCL expressions e. $c(e)$ can be considered a count of the number of basic semantic elements in a specification (identifiers plus composite expressions). We also include a token count measure $t(e)$, which is used for clone detection. We will investigate how LOC correlates with the c measure of size.

Table 1. OCL expression complexity measures

Expression e	Complexity $c(e)$	Token count $t(e)$
Numeric, boolean or String value	0	1
Identifier *iden* Basic expression $obj.f$	1 $c(obj) + c(f) + 1$	1 $t(obj) + t(f) + 1$
Operation call $e(p1, ..., pn)$	$c(e) + 1 + \Sigma_i c(pi)$	$t(e) + n + 1 + \Sigma_i t(pi)$
Unary expression $op\ e$ $e \rightarrow op()$	$1 + c(e)$	$1 + t(e)$ $4 + t(e)$
Binary expression $e1\ op\ e2$ $e1 \rightarrow op(e2)$	$c(e1) + c(e2) + 1$	$t(e1) + t(e2) + 1$ $t(e1) + t(e2) + 4$
Ternary expression $op(e1, e2, e3)$ *if* $e1$ *then* $e2$ *else* $e3$ *endif*	$c(e1) + c(e2) + c(e3) + 1$	$t(e1) + t(e2) + t(e3) + 5$ $t(e1) + t(e2) + t(e3) + 4$
$Set\{e1, ..., en\}$ $Sequence\{e1, ..., en\}$	$1 + \Sigma_i c(ei)$	$2 + n + \Sigma_i t(ei)$

A similar measure can be given to activities (Table 2 shows the values for UML-RSDS syntax, similar definitions can be given for the ATL and ETL statement syntax).

Table 2. Activity complexity measures

Activity s	Complexity $c(s)$	Token count
`return` e	$1 + c(e)$	$1 + t(e)$
$v := e$	$c(v) + c(e) + 1$	$t(v) + t(e) + 1$
$s1;\ s2$	$c(s1) + c(s2) + 1$	$t(s1) + t(s2) + 1$
Operation call $e(p1, ..., pn)$	$c(e) + 1 + \Sigma_i c(pi)$	$t(e) + n + 1 + \Sigma_i t(pi)$
`if` e `then` $s1$ `else` $s2$	$1 + c(e) + c(s1) + c(s2)$	$3 + t(e) + t(s1) + t(s2)$
`for` $v : e$ `do` s	$c(e) + c(s) + 1$	$3 + t(e) + t(v) + t(s)$
`while` e `do` s	$c(e) + c(s) + 1$	$t(e) + t(s) + 2$
`break`	1	1
`continue`	1	1

Using these measures, $c(r)$ for a transformation rule r is taken as the sum of the c measures of its parts (such as *from*, *to* and *do* clauses in ATL), likewise for operation definitions. The semantic complexity $c(\tau)$ of a transformation is taken as the sum of the complexities of its rules and operations. We also adopt the metric of *fan-out* from [9], this is the number of different rules or operations called from one rule or operation. This quantity has a direct impact on the understandability of the calling rule/operation.

We also consider LOC measures of size because this is widely used for TD estimation. We will evaluate flaw density both wrt LOC and complexity. Based on [9], we adopt 50 LOC per rule/operation and 500 LOC per transformation as size thresholds, for size measured by LOC. These thresholds apply to ATL, ETL and QVT-R. For UML-RSDS we adopt limits based on expression complexity (100 and 1000 respectively) since UML-RSDS specifications consist of graphical use cases and class diagrams. These limits are based on our experience with maintenance of UML-RSDS transformations. In future work we will evaluate the validity of these limits using normalisation of encountered values [14].

Technical debt in MT developments will therefore be measured by identifying the frequency of occurrence of the following specific 'bad smells' in MT specifications:

ETS: Excessive transformation size ($c(\tau) > 1000$, or length > 500 LOC)
ENR: Excessive number of rules ($nrules > 10$)
ENO: Excessive number of helpers/operations ($nops > 10$)
UEX: Excessive use of undefined execution orders/priorities between rules (>10 undefined orderings)
ERS: Excessive rule size ($c(r) > 100$ or length greater than 50 LOC)

EHS: Excessive helper size ($c(h) > 100$ or length > 50 LOC)
EPL: Excessive parameter list (for transformation, rules, and helpers): >10 parameters including auxiliary rule/operation variables
DC: Duplicate expressions/code (duplicate expressions or statements x with token count $t(x) > 10$)
CC: Cyclomatic complexity (of rule logic or of procedural code) (>10)
CBR: Coupling between rules (number of rule/operation explicit or implicit calling relations $> nrules + nops$, or any cyclic dependencies exist in the rule/operation call graph).
EFO: Excessive fan-out of a rule/operation (>5 different rules/operations called from one rule/operation).

Number of tokens is used for detecting clones, because in this case value expressions should be counted as contributing to the clone. The lower limit for clone size is set to avoid trivial clones. It could be reduced, at the cost of increased processing time. In [15] clones of any size are considered. In [7], a lower bound of 50 tokens is used for code clone detection. We experimented with using 25 tokens as the threshold, but this led to many significant clones being ignored, and we adopted 10 tokens for our analysis. Only identical clones are counted.

At present, we limit our scope to considering individual transformations, rather than transformations in a system of inter-operating transformations. We also do not consider problematic issues in the use of OCL [6] – OCL 'smells' such as the use of chained *implies*, 'magic literals', chained *forAll* quantifiers, long chained navigations in expressions, and other constructions which impair the comprehensibility of the specification.

3 Analysis and Results

The measures of TD are computed on the abstract syntax representations of ATL, ETL, QVT-R and UML-RSDS specifications, according to the respective metamodels of these languages. The languages have many similarities at this level (eg., top-level rules in QVT-R correspond to non-lazy rules in ETL, non-lazy non-called rules in ATL, and to use case constraints in UML-RSDS). Hence the same general specification of measures can be applied to each language, with some differences to account for the different language styles and semantics.

We present the results in Tables 3, 4, 5, 6, 7 and 8. For *ETS* we show separately the LOC measures *rs* of the transformation rules and *os* of the helper operations, after their total. *ENR* is the number of rules in the case, *ENO* is the number of operations. *ERS* is the number of rules with length over the threshold (50 LOC), likewise *EHS* for operations. *EPL* is the number of rules/operations with more than 10 parameters, including local auxiliary variables. *EFO* is the number of rules/operations which depend on more than 5 rules/operations. *CC* is the number of rules/operations over the *CC* threshold (10). *CBR* is expressed as $CBR_1(CBR_2)$ where CBR_1 is the total number of rule/operation dependencies, and CBR_2 is the number of rules/operations which occur in cycles of calling dependencies. *DC* is the number of distinct cloned expressions (e with $t(e) > 10$) in the case. Underlined measures in Tables 3 and 6 identify where flaws occur.

3.1 ATL

For ATL we consider the cases of Table 3 from the ATL transformations zoo, which is widely used in surveys of model transformations. The cases are chosen as being typical of medium to large sized ATL transformations.

For ATL, *UEX* is $n * (n - 1)/2$ where n is the number of concrete non-lazy, non-called rules. For all of the ATL examples *EPL* and *EFO* are 0, so are omitted. Where a transformation consists of several subtransformations, we list these as (i), (ii) etc. below the main transformation entry.

Table 3. Technical debt measures for ATL

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*CC*	*CBR*	*DC*	*UEX*
MOF to UML	935 (746, 189)	11	11	5	0	0	27(0)	7	55
KM3 to DOT	451 (251,200)	7	18	1	0	0	33(0)	4	21
MySQL to KM3	995 (571, 424)	20	28	1	0	1	62(4)	7	71
(i) *XML2XML*	101 (87, 14)	4	1	0	0	0	2(0)	2	6
(ii) *XML2MySQL*	281 (137,144)	5	10	0	0	0	22(2)	2	10
(iii) *MySQL2KM3*	613 (347,266)	11	17	1	0	1	38(2)	3	55
Excel Injector	395 (231,164)	11	10	0	0	0	38(0)	3	55
Excel Extractor	311 (251,60)	13	5	0	0	0	6(1)	2	66
(i) *SpreadsheetML Simplified2XML*	263 (246,17)	12	1	0	0	0	1(0)	2	66
(ii) *XML2ExcelText*	48 (5,43)	1	4	0	0	0	5(1)	0	0
PetriNet to/from PathExpression	1267 (799,468)	23	32	2	1	0	88(2)	8	47
(i) *PetriNet2PathExp*	70 (70,0)	3	0	0	0	0	0(0)	1	3
(ii) *XML2PetriNet*	228 (136,92)	5	8	0	0	0	22(0)	2	10
(iii) *PetriNet2XML*	222 (189,33)	5	3	1	0	0	12(0)	4	10
(iv) *PathExp2PetriNet*	104 (87,17)	3	1	0	0	0	5(0)	0	3
(v) *TextualPathExp2PathExp*	643 (317,326)	7	20	1	1	0	49(2)	1	21
Make to Ant	368 (242,126)	13	11	0	0	0	13(2)	2	31
(i) *XML2Make*	147 (73,74)	5	7	0	0	0	7(1)	0	10
(ii) *Ant2XML*	177 (164,13)	7	1	0	0	0	2(0)	2	21
(iii) *XML2Text*	44 (5,39)	1	3	0	0	0	4(1)	0	0
Maven to Ant	1307 (1139,168)	90	18	0	0	0	80(0)	7	1326
(i) *XML2Maven*	575 (472,103)	36	13	0	0	0	74(0)	3	630
(ii) *Maven2Ant*	360 (308,52)	30	4	0	0	0	4(0)	1	420
(iii) *Ant2XML*	372 (359,13)	24	1	0	0	0	2(0)	3	276

Table 4 gives a summary of the technical debt of these cases. To compute the number of flaws in a transformation, we count 1 for each of *ETS*, *ENR*, *ENO*, *UEX*, CBR_1 over the thresholds, plus *ERS* + *EHS* + *CC* + *EPL* + *EFO* + *DC* + CBR_2. For a transformation system, we sum the number of flaws in each of its subtransformations. We use the transformation intent classifications of [12] for MT categories. It is noticeable that the number of flaws per LOC is quite similar across all of the cases, (the standard deviation is 0.0023).

It can be noted that the ratio of complexity to LOC is 1.71, reflecting the relatively low semantic density of typical ATL specifications. The flaw rate per semantic element is 0.00931 (number of flaws divided by complexity).

Table 4. Results summary for ATL

Transformation	Category [12]	LOC	$c(\tau)$	% in rules	# flaws	Flaws/LOC
MOF to UML	Migration	935	1002	79.7%	17	0.018
KM3 to DOT	Refinement	451	926	55.6%	8	0.017
MySQL to KM3	Abstraction	995	1726	57.3%	19	0.019
Excel Injector	Migration	395	601	58.5%	6	0.015
Excel Extractor	Migration	311	528	81%	5	0.016
Petri Net from/to	Semantic map	1267	1645	63%	20	0.016
Make to Ant	Migration	368	808	65.7%	5	0.013
Maven to Ant	Migration	1307	3075	87%	16	0.012
Average		753.6	1288.9	70.2%	12	0.016

3.2 ETL

ETL has a similar rule and transformation structure to ATL, but with a more general processing model and more complex semantics. For ETL we define UEX as $\frac{n*(n-1)}{2}$ where n is the number of concrete non-lazy rules. We identified ETL cases to analyse from the Eclipse ETL repository (git.eclipse.org), and from other published cases (github.com/epsilonlabs).

ETL has implicit invocation of rules by rules or operations, where the text of the transformation does not contain an explicit reference to rules that may be invoked due to *equivalent*/*equivalents* expressions. In calculating the call graph and *CBR* metric, such implicit calls must be taken into account. In ETL, an expression $e.equivalent()$ may implicitly invoke any concrete lazy or non-lazy rule which has an input variable $v : T$ with T containing the actual value of e *at runtime*. Thus the calling rule or operation implicitly depends upon all concrete rules in the transformation, potentially leading to large values for fan-out and call graph size. The abbreviated form $v ::= e$ of $v = e.equivalent()$ is considered in the same manner. The detailed TD evaluations for ETL may be found at nms.kcl.ac.uk/kevin.lano/icmt18.pdf.

Table 5 gives a summary of the technical debt of these cases. The same computation of number of flaws is used as for ATL. It is noticeable that the rate of flaws per LOC is higher than for ATL in general, and with a much wider range of rates than for ATL (the s.d. is 0.06). This may be due to the wide variety of styles supported by ETL, from the highly imperative transformations of *StateElimination*, to the very implicit and declarative *CopyOO*. Using the F-distribution test, there is a statistically-significant difference between the ETL and ATL TD levels [17]. In the most complex cases, such as *MDDTIF*, three forms of inter-rule/operation dependence are used simultaneously: inheritance, explicit calls and implicit calls, leading to high values for *CBR* and *EFO*.

From Table 5 we have that complexity/LOC for ETL is 2.9, indicating a greater semantic density in ETL specifications than for ATL. The rate of flaws per semantic element is 0.024.

Table 5. Results summary for ETL

Transformation	Category	LOC	$c(\tau)$	% in rules	# flaws	Flaws/LOC
Flowchart2HTML	Code-generation	163	377	100%	2	0.012
CopyFlowchart	Migration	57	153	100%	7	0.122
CopyOO	Migration	110	438	100%	23	0.209
In2out	Migration	19	53	100%	1	0.052
OO2DB	Refinement	142	464	85.2%	6	0.042
RSS2ATOM	Refinement	88	154	84%	6	0.068
Tree2Graph	Refinement	15	37	100%	1	0.066
uml2xsd	Migration	17	44	100%	1	0.058
MDDTIF	Refinement	145	377	95.8%	26	0.179
Argouml2ecore	Migration	96	321	79%	13	0.135
StateElimination	Refactoring	313	1062	49.5%	7	0.022
TTC Live Case 2017	Refinement	206	573	79%	6	0.029
uml2Simulink	Refinement	148	477	77%	11	0.074
Average		116.8	348.46	80.5%	8.46	0.072

3.3 QVT-R

For QVT-R transformations the *CBR* and *UEX* measures are of particular interest, since QVT-R rules (termed 'relations') may be interdependent in several different ways: a rule may refer to another in its *when* or *where* clause, and may have a recursive dependency upon itself, and may override another rule. *UEX* is taken as $\frac{n*(n-1)}{2}$ where n is the number of concrete top-level rules in a transformation. A special feature of QVT-R is that relations may define a large number of auxiliary variables to transfer data from one relation domain to another, or to transfer data between relations. This may result in high *EPL* values even for small transformations. This can cause problems in understanding the relations because the meaning and role of each variable needs to be understood.

The OCL syntax used in QVT-R differs from that of the other MT languages. We evaluate complexity directly on this syntax, rather than upon its standard OCL translation. Thus an object specification e

```
obj : E1 { att = var, rel = obj2 : E2{} }
```

has $c(e) = 11$, versus 19 for its conventional OCL equivalent expression:

$$obj : E1 \ and \ obj.att = var \ and \ obj2 : E2 \ and \ obj.rel = obj2$$

We have selected published examples of QVT-R specifications from the ModelMorf repository, from the QVT-R standard, and from published papers [13]. Table 6 gives the measures for the selected QVT-R cases.

Table 6. Technical debt measures for QVT-R

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EPL*	*EFO*	*CBR*	*DC*	*UEX*
HierarchicalStateMachine2 FlatStateMachine	85 (79, 6)	3	1	0	0	1	0	3(0)	0	3
AbstractToConcrete	47 (47,0)	1	0	0	0	0	0	0(0)	0	0
ClassModelToClassModel	85 (85,0)	3	0	0	0	0	0	4(1)	0	1
DNF	396 (396,0)	9	0	4	0	4	0	10(4)	3	6
DNF_bbox	263 (263,0)	5	0	4	0	5	0	4(0)	3	6
SeqToStm	104 (104,0)	4	0	0	0	1	0	4(0)	0	6
seqtostmct	149 (149,0)	5	0	0	0	0	0	6(3)	0	0
UmlToRdbms	238 (226,12)	7	1	1	0	1	0	10(3)	0	3
UmlToRel	98 (65,33)	2	2	0	0	0	0	3(0)	0	1
RelToCore	2038 (1937, 101)	50	5	11	0	13	5	141(7)	3	15
Bpmn2UseCase	522 (522,0)	23	0	0	0	0	0	12(0)	4	55
hsm2nhdm (recursion)	48 (48,0)	5	0	0	0	0	0	5(2)	0	3

Table 7 gives a summary of the technical debt of these cases. The same computation of number of flaws is used as for ATL. There are 0.023 flaws/LOC and 0.011 flaws per semantic element, figures intermediate between ATL and ETL. There are 2.09 semantic elements/LOC, a density figure again intermediate between ATL and ETL.

Table 7. Results summary for QVT-R

Transformation	Category	LOC	$c(\tau)$	% in rules	# flaws	Flaws/LOC
HSM2FlatSM	Abstraction	85	137	93%	1	0.011
AbstractToConcrete	Refactoring	47	57	100%	0	0
ClassModelToClassModel	Migration	85	85	100%	2	0.023
DNF	Refactoring	396	665	100%	16	0.04
DNF_bbox	Refactoring	263	470	100%	12	0.045
SeqToStm	Refinement	104	175	100%	1	0.009
seqtostmct	Refinement	149	162	100%	4	0.027
UmlToRdbms	Refinement	238	314	95%	6	0.025
UmlToRel	Refinement	98	75	95%	0	0
RelToCore	Refinement	2038	5415	95%	43	0.021
Bpmn2UseCase	Migration	522	877	100%	7	0.013
hsm2nhdm (recursion)	Abstraction	48	105	100%	2	0.041
Average		339.4	711.25	96%	7.83	0.023

3.4 UML-RSDS

For UML-RSDS transformations we consider three substantial case studies: two parts of the UML2C code generator [11] and the class diagram modulariser *cra* from [10]. A range of other examples are also included from nms.kcl.ac.uk/kevin.lano/uml2web/zoo. In total there are 36 individual transformations and 10 transformation systems. The TD detailed measures for UML-RSDS are available at nms.kcl.ac.uk/kevin.lano/icmt18.pdf.

Table 8 summarises the results for UML-RSDS. We estimated LOC by printing the specification files and counting lines of operation and use case code, omitting metamodel class, generalisation and association declarations.

Table 8. Results summary for UML-RSDS

Transformation	Category	LOC	$c(\tau)$	% in rules	# flaws	Flaws/LOC
uml2Ca	Code generation	874	1272	69%	22	0.025
uml2Cb	Code generation	1576	5621	16%	119	0.075
cra	Refactoring	490	1360	32%	12	0.024
f2p/p2f	Bidirectional	58	158	86%	3	0.052
calc	Analysis	15	83	100%	0	0
movies	Analysis	156	432	40%	3	0.019
Monte-Carlo sim	Analysis	51	90	68%	0	0
Nelson-Seigal	Refinement	458	1219	67%	15	0.032
CDO	Analysis	94	182	17%	2	0.02
PetriNet to SM	Refactoring	66	174	100%	0	0
Average		383.8	1059.1	34.9%	17.6	0.0458

It can be noted that the $c(\tau)$ measure is around 2.76 times the LOC, a similar level of semantic density to ETL.

An interesting aspect of the results is the balance of functionality between helpers and rules. Excessive use of helpers produces transformations which are akin to programs in a functional programming language. In the largest transformation (*uml2Cb*, *cra*) there is a considerable imbalance of functionality towards helpers, whilst smaller transformations such as the Monte-Carlo simulator are more balanced.

4 Discussion and Summary of Results

We consider the results for each language with respect to the research questions. For ATL, for **RQ1**, all of the 19 individual transformations had flaws (100%), and 8 of 8 transformation systems contained transformations with flaws (100%). For **RQ2**, the most common flaws were DC (15/19), CBR – either $CBR_1 > 0$ or $CBR_2 > 0$ – (13/19), UEX (10/19), ENR (7/19) and ENO (5/19).

A particular issue in ATL is the use of *resolveTemp* expressions in rules to look up target model elements produced by another rule, during transformation processing. This is considered a semantic complexity factor in [2] because it introduces a syntactic and semantic dependency of the rule calling *resolveTemp* upon the rule identified by the call. We include the rule-to-rule dependencies induced by *resolveTemp* in the CBR measure.

For ETL, the critical factor in the considered transformations is the implicit *CBR* due to usage of *equivalent* and related operators. For **RQ1**, 19 of the 24 individual transformations contained flaws (79%), and all of the 13 transformation systems contained transformations with flaws (100%). For **RQ2** the most common flaws were *CBR* (18/24), *EFO* (7/24) and *DC* and *UEX* (both 5/24). Excessive size of rules/helpers or transformations was not a significant problem.

For QVT-R, for **RQ1**, out of 12 transformations, 10 had flaws (83%). For **RQ2**, *EPL* and *CBR* both occur in 6 of 12 transformations, whilst *DC* and *ERS* occur in 4. High values of *EPL* arise because of the use of many local variables within QVT-R relations, to facilitate bidirectional use of the relations. *CBR* flaws arise from the unstructured nature of QVT-R transformations in which rules may be closely inter-dependent. In the largest transformation, *relToCore*, there is informal stratification of the transformation into groups of rules, but this could be clearer if the transformation were explicitly decomposed into client and supplier sub-transformations.

For UML-RSDS, for **RQ1**, out of 36 transformations, 16 had some flaws (44%), whilst 7 of 10 transformation systems contained some transformations with flaws (70%). The *uml2Cb* case somewhat distorts the flaw density data: without this case the flaws per LOC would be the same as for QVT-R.

For **RQ2**, excessive *CBR* occurs in 9 transformations. *DC* also occurs in 9 cases. *ENO* occurs in 7 cases. *CC* occurs in 6 cases. In all cases, the coupling issues concern complex dependencies between helpers, rather than between rules. The prevalence of *CBR* and *ENO* flaws suggest overuse of helpers/operations. Poor structure and high numbers of flaws were apparent in the largest transformations.

For **RQ3**, Table 9 summarises the different prevalence of TD types in different MT languages, counting the number of individual transformations which have flaws of each kind. Unusual patterns of TD are emphasised.

In summary, it seems that excessive *CBR* and *DC* are the most significant design flaws which arise across all MT languages, although there are significant variations in the kinds of TD problem between different languages. These findings suggest that an important factor in understanding and maintaining model transformations are the dependencies between rules/operations.

CBR could be reduced by the stratification and modularisation of transformations into smaller units. Currently MT languages offer such *external* composition [16] of transformations by the sequencing of individual transformations: a facility heavily used in the UML-RSDS examples in particular. However it seems what is needed is a modularisation mechanism to support a hierarchical client-supplier relationship between transformations, with the internal details of the supplier module independent of its clients. This would enable, for example, a transformation mapping OCL expressions to be called as a 'black box' from a transformation mapping UML activities. The combination of these two transformation processes into *uml2Cb* is a significant factor in its high flaw count.

Table 9. Technical debt prevalence in different MT languages

TD category	ATL	ETL	QVT-R	UML-RSDS	Overall
CBR	13/19	18/24	6/12	9/36	46/91
DC	15/19	5/24	4/12	9/36	33/91
UEX	10/19	5/24	2/12	0/36	17/91
ENR	7/19	0/24	2/12	3/36	12/91
ENO	5/19	0/24	0/12	***7/36***	12/91
ERS	5/19	2/24	4/12	0/36	11/91
EFO	0/19	***7/24***	1/12	1/36	9/91
EPL	0/19	2/24	***6/12***	0/36	8/91
ETS	4/19	0/24	2/12	2/36	8/91
CC	1/19	0/24	0/12	***6/36***	7/91
EHS	1/19	2/24	0/12	1/36	4/91

Table 10 shows the overall figures for LOC, c, and flaws, for each language.

Table 10. Overall size and TD results

Language	LOC	c	c/LOC	Flaws	Flaws/LOC	Flaws/c
ATL	6029	10311	1.71	96	0.016	0.009
ETL	1519	4530	2.98	110	0.072	0.024
QVT-R	4073	8535	2.09	94	0.023	0.011
UML-RSDS	3838	10591	2.76	176	0.046	0.017
Overall	15459	33967	2.19	476	0.031	0.014

The flaw density figures for ETL and UML-RSDS are higher than for ATL and QVT-R, both wrt LOC and wrt c. This difference can be due to specific language features such as implicit calls (ETL), or excessive use of operations (UML-RSDS), but also due to the use of ETL and UML-RSDS for more complex transformations, including update-in-place cases such as *PetriNet to SM* which would be very difficult to express in ATL or QVT-R.

We can also compare the levels of TD in different categories of transformation, across languages. Table 11 shows the TD frequency for the main categories of transformations in our survey. Although the sample numbers are too small for statistical significance, the difference in flaw levels between the main categories is in accord with expectations that more complex MT tasks such as refinement will result in transformations with higher numbers of flaws compared to simpler tasks such as migration.

For **RQ4**, TD densities in developer-coded Eclipse projects have been measured in [7], with values ranging from 0.005 to 0.04 flaws per LOC, with an

Table 11. TD for MT categories

Category	LOC	Flaws	Flaws/LOC
Code generation	2613	143	0.055
Bidirectional	58	3	0.052
Refinement	4280	133	0.031
Refactoring	1575	47	0.029
Migration	4222	103	0.024
Abstraction	1128	22	0.019
Analysis	316	5	0.016
Semantic map	1267	20	0.016

average around 0.015. We also evaluated manually coded versions of a UML to C++ translator (18,100 lines of Java), and 2 versions of the CDO case study (200 lines of C++, and 236 lines of Java) using the PMD code size library (https://pmd.github.io). These had TD levels of 0.009/LOC, 0.021/LOC and 0.017/LOC, respectively. The TD levels of ETL and UML-RSDS are high in comparison with these code TD results, whilst ATL and QVT-R exhibit TD levels more typical of executable code.

5 Threats to Validity

The conclusions we have drawn may be challenged on the basis that (a) the measures chosen are not appropriate for evaluating TD; (b) the selection of transformation cases was unrepresentative; (c) the basis of TD measurement of different MT languages are not equivalent.

Regarding (a), we have adopted established TD measures which have been used extensively for TD evaluation of programs. We have used 500 LOC as a threshold for transformation size, and 50 LOC as a threshold for rule/operation size. This is partly justified by the fact that overall the ratio of complexity to LOC is close to 2, and thus the 50/500 LOC limits correspond, on average, to the 100/1000 limits for complexity. In addition, out of 74 cases where both transformation LOC and $c(\tau)$ were available, in 69 cases (93%) the thresholds were in agreement: both $c(\tau) > 1000$ and $LOC > 500$ in 9 cases, or both $c(\tau) \leq 1000$ and $LOC \leq 500$ in 60 cases. Two cases were over 500 LOC but below 1000 $c(\tau)$ whilst 3 had the converse. Transformations that operate on large metamodels or that perform complex tasks will typically have high TD if they are not effectively modularised (such as MOF to UML, RelToCore, or uml2Cb). Decomposition into subtransformations (as for PetriNet to/from PathExpression, and cra) can significantly reduce TD levels, even for transformations with large metamodels/complex tasks.

Regarding (b), we have considered public repositories of cases and published examples of MT specifications for each language, and the selection of cases has been on the same basis for each language. For each language, we have endeavoured to obtain a wide range of transformation examples, spanning in size from small cases to the largest cases available, and across the range of all available categories of transformation. However, higher TD measures were obtained for languages (ETL, UML-RSDS) with a wider range of transformation facilities, and hence that have been applied to more complex tasks. It can be noted that the ETL cases are significantly smaller (average complexity size 348) than the ATL, QVT-R or UML-RSDS cases (average sizes 1289, 711 and 1059). There are few large publicly-available ETL cases, which restricted our choice for analysis.

Regarding (c), some distortion is introduced by the analysis of cases where one MT language feature is used to express another concept in the source specification. For example, in the KM32DOT ATL transformation, the first 9 helper operations *DiagramType*(), *Mode*(), etc. are used to represent the parameters of the transformation. Such cases would require manual correction in the analysis, but we consider that it is preferable to analyse the transformations on the basis of their actual text, not on the basis of how the specifier intended the text to be interpreted (since this knowledge may not be available in some cases, leading to inconsistency in the analysis).

6 Related Work

One of the first works to consider metrics for MT was [9]. They define measures for the size and complexity of QVT-R transformations, including lines of code, number of relations (corresponding to number of rules), and specific measures for the size and inter-relationship of QVT-R rules. Their analysis is limited to QVT-R and does not consider clone detection or detailed analysis of the rule dependency graph. They evaluated one large (auto-generated) QVT-R transformation and three moderate/small transformations. Undefined execution order between rules is a significant problem in the large transformation. In [1], measures of ATL and QVT-R and QVT-O are computed for versions of two transformations in each language. In [2], seven ATL transformations are evaluated by metrics and by expert analysis, in order to identify correlations between metric values and expert evaluation of quality characteristics. Wimmer et al. [18] use quality measures to evaluate the effect of MT refactorings. They adopt ERS, DC and EFO as quality criteria for ATL transformations.

Clone detection in transformations is considered by [15], and they evaluate alternative tools for clone detection in graph transformations.

7 Conclusion

We have shown that technical debt can be evaluated for different MT languages. We have evaluated 91 transformations in four transformation languages, and identified significant differences between the languages in their frequency and type of TD: while ATL and QVT-R cases have flaw densities similar to traditional code, the more complex languages ETL and UML-RSDS have cases with typically higher flaw densities. All languages suffer from flaws due to complex dependencies between rules/operations. This may be a symptom of poor modularisation facilities in MT languages. The identification of design flaws can help MT specifiers to improve their transformations and to prioritise refactoring or other quality improvement work on their transformations.

References

1. van Amstel, M., Bosems, S., Kurtev, I., Ferreira Pires, L.: Performance in model transformations: experiments with ATL and QVT. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 198–212. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21732-6_14
2. van Amstel, M., van den Brand, M.: Using metrics for assessing the quality of ATL model transformations. In: MtATL 2011 (2011)
3. Arendt, T., Taentzer, G.: UML model smells and model refactorings in early software development phases. Technical report FB 12. Philipps Universitat, Marburg (2010)
4. Basili, V.: Software modeling and measurement: the goal/question/metric paradigm (1992)
5. Batot, E., Sahraoui, H., Syriani, E., Molins, P., Sboui, W.: Systematic mapping study of model transformations for concrete problems. In: Modelsward 2016, pp. 176–183 (2016)
6. Correa, A., Werner, C.: Refactoring OCL specifications. SoSyM **6**, 113–138 (2007)
7. He, X., Avgeriou, P., Liang, P., Li, Z.: Technical debt in MDE: a case study on GMF/EMF-based projects. In: MODELS 2016 (2016)
8. IEC/ISO, 25010 Systems and software engineering - systems and software quality models (2011)
9. Kapová, L., Goldschmidt, T., Becker, S., Henss, J.: Evaluating maintainability with code metrics for model-to-model transformations. In: Heineman, G.T., Kofron, J., Plasil, F. (eds.) QoSA 2010. LNCS, vol. 6093, pp. 151–166. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13821-8_12
10. Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S.: Solving the CRA case using UML-RSDS. In: TTC 2016 (2016)
11. Lano, K., et al.: Translating from UML-RSDS OCL to ANSI C. In: OCL 2017 (2017)
12. Lucio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Model transformation intents and their properties. SoSyM **15**, 647–684 (2016)
13. Macedo, N., Cunha, A.: Least-change bidirectional model transformation with QVT-R and ATL. SoSyM **15**, 783–810 (2016)
14. Marinescu, R.: Assessing technical debt by identifying design flaws in software systems. IBM J. Res. Dev. **56**(5), 9:1–9:13 (2012)
15. Struber, D., Ploger, J., Acretoaie, V.: Clone detection for graph-based MT languages. In: ICMT 2016 (2016)
16. Wagelaar, D.: Composition techniques for rule-based MT languages. In: ICMT 2008 (2008)
17. Weatherill, G.B.: Elementary Statistical Methods. Chapman and Hall, London (1978)
18. Wimmer, M., et al.: A Catalogue of Refactorings for model-to-model transformations. J. Object Technol. **11**(2), 1–40 (2012)

CoqTL: An Internal DSL for Model Transformation in Coq

Massimo Tisi[1(✉)] and Zheng Cheng[2]

[1] IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France
massimo.tisi@imt-atlantique.fr
[2] Research Center INRIA Rennes - Bretagne Atlantique, Rennes, France
zheng.cheng@inria.fr

Abstract. In model-driven engineering, model transformation (MT) verification is essential for reliably producing software artifacts. While recent advancements have enabled automatic Hoare-style verification for non-trivial MTs, there are certain verification tasks (e.g. induction) that are intrinsically difficult to automate. Existing tools that aim at simplifying the interactive verification of MTs typically translate the MT specification (e.g. in ATL) and properties to prove (e.g. in OCL) into an interactive theorem prover. However, since the MT specification and proof phases happen in separate languages, the proof developer needs a deep knowledge of the translation logic. Naturally any error in the MT translation could cause unsound verification, i.e. the MT executed in the original environment may have different semantics from the verified MT.

We propose an alternative solution by designing and implementing an internal domain specific language, namely CoqTL, for the specification of declarative MTs directly in the Coq interactive theorem prover. Expressions in CoqTL are written in Gallina (the specification language of Coq), increasing the possibilities of reuse of native Coq libraries in the transformation definition and proof. In this paper we introduce CoqTL, we evaluate its practical applicability on a case study, and identify its limitations.

Keywords: Model-driven engineering · Model transformation
Interactive theorem proving · Coq

1 Introduction

Model-driven engineering (MDE), i.e. software engineering centered on software models and MTs, is widely recognized as an effective way to manage the complexity of software development. With the increasing complexity of MTs (e.g., in automotive industry [20], medical data processing [22], aviation [2]), it is urgent to develop techniques and tools that prevent incorrect MTs from generating faulty models. The effects of such faulty models could be unpredictably propagated into subsequent MDE steps, e.g. code generation.

Deductive verification is a promising approach for quality assurance in MT: *correctness* is specified by MT developers using contracts (i.e. pre/postconditions), then the semantics of the MT language together with contracts and metamodels are encoded into a deductive theorem prover. Thanks especially to recent advancements in SMT solvers, automatic deductive verification is giving good results in several scenarios [3,4,6,16]. However, because of the general undecidability, interactive deductive verification is inevitable for complex tasks (for instance, automatic deductive theorem provers usually lack support for induction or finding witnesses for existential quantifiers).

Coq is an interactive theorem prover. The user can use Coq to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of proofs (in the sense that routine proofs can be automatically performed while difficult proofs require human guidance). It has been used to prove non-trivial mathematical theorems, or as an environment for developing formally certified software and hardware (e.g. [10,15]). While not strictly needed for understanding this paper, we refer the reader to [18] for an introduction to the Coq system.

Previous work aiming at simplifying the interactive verification of MTs, has already proposed translations from MT specifications (e.g. in MT languages like ATL) and properties to prove (e.g. in OCL) into Coq. However, the practical applicability of this translational approach is hampered by the fact that the two phases of MT specification and correctness proof require developments in languages (e.g. ATL+OCL and Coq, respectively) at two different levels of abstraction. The proof developer needs a deep knowledge of the translation logic to be able to write meaningful proofs. Any change in the MT code propagates through the translator, and it is difficult to predict the proof steps that will be invalidated. Naturally any error in the MT translation could cause unsound verification, i.e. the MT executed in the original environment may have different semantics from the verified MT. Certifying that the semantics of the MT language has being correctly axiomatized in the back-end theorem prover is a hard task, and very few attempts exist [1,6].

Coq includes Gallina, a functional programming language with pattern matching and rich type system, well suited as a platform for embedding domain-specific programming languages (DSLs) (e.g. [7]). In this work, we draw on this aspect of Coq and propose a DSL, namely CoqTL, to turn Coq into a tool for developing certified MTs. We argue that using an internal DSL for the MT specification phase simplifies the iterative process of MT development and proof in MDE. Moreover, expressions in CoqTL are directly written in Gallina, increasing the possibilities of reuse of sophisticated native Coq libraries during the transformation definition and proof.

Our main contributions are:

- We design and implement CoqTL, to our knowledge the first DSL for rule-based MT in Coq (Sect. 3.2). The language is both functional and declarative in style, its syntax and semantics is inspired from ATL [12] (hence it should be familiar also to users of other rule-based MT languages, like ETL [13], or

RubyTL [8]). Thus, CoqTL aims to lighten the cognitive load of MT developers trying to build certified MTs in Coq.
– We design and implement a transformation engine in Coq that interprets programs written in CoqTL to transform models (Sect. 3.3). The engine includes an on-the-fly parser that transforms the domain-specific syntax into a Coq data structure to interpret. The parser is transparently invoked (by an extensive use of the Coq Notation mechanism) so that any Coq development environment is able to support the domain-specific CoqTL syntax without requiring ad-hoc modifications.
– We show the practical applicability of CoqTL, by using it to specify a sample transformation, prove non-trivial contracts over it and automatically extract a certified implementation.

We make CoqTL publicly available as open source[1]. The repository contains also the example and proofs described in this paper.

Paper Organization. We motivate our work by a sample transformation in Sect. 2. Section 3 illustrates the design of CoqTL in detail. In Sect. 4 we prove theorems on a CoqTL specification. Section 5 compares our work with related research, and Sect. 6 draws conclusions and lines for future work.

2 Class to Relational in CoqTL

We consider a very simplified version of the transformation from class diagrams to relational schemas (arguably, the Hello World transformation in the MT community). The example is intentionally very small, so that it can be completely illustrated within this paper. However we believe it to be easily generalizable by the reader to more complex scenarios. The structure of the involved metamodels is shown in Fig. 1.

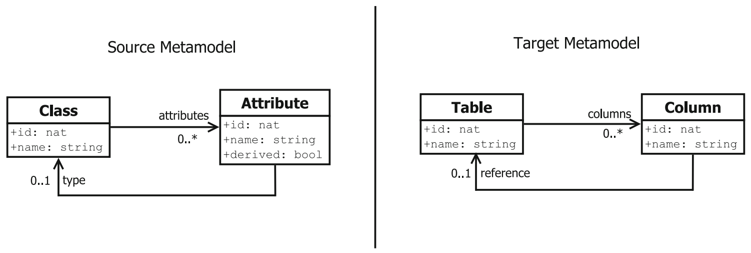


Fig. 1. A simplified structural metamodel for class diagrams (left), and relational schemas (right)

The left part of Fig. 1 shows the simplified structural metamodel of class diagrams. Each class diagram contains a list of named classes with identities.

[1] CoqTL (online). https://github.com/atlanmod/CoqTL.

```
1  Definition Class2Relational :=
2    transformation from ClassMetamodel to RelationalMetamodel
3      with m as ClassModel := [
4
5      rule Class2Table
6        from
7          element c class Class from ClassMetamodel
8            when true
9        to
10         [
11           output "tab"
12             element t class Table from RelationalMetamodel :=
13               BuildTable (getClassId c) (getClassName c)
14             links
15               [
16                reference TableColumns from RelationalMetamodel :=
17                  attrs ← getClassAttributes c m;
18                  cols ← resolveAll (match Class2Relational m) "col" Column (singletons attrs);
19                  return BuildTableColumns t cols
20               ]
21         ];
22
23     rule Attribute2Column
24       from
25         element a class Attribute from ClassMetamodel
26           when (negb (getAttributeDerived a))
27       to
28         [
29           output "col"
30             element c class Column from RelationalMetamodel :=
31               BuildColumn (getAttributeId a) (getAttributeName a)
32             links
33               [
34                 reference ColumnReference from RelationalMetamodel :=
35                   cl ← getAttributeType a m;
36                   tb ← resolve (match Class2Relational m) "tab" Table [cl];
37                   return BuildColumnReference c tb
38               ]
39         ]
40   ].
```

Listing 1.1. Class2Relational model transformation in CoqTL

Each class contains a list of named and typed attributes with unique identities. In this simplified model we do not consider attribute multiplicity (i.e., all attributes are single-valued). Primitive data types are not explicitly modeled, thus we consider every attribute without an associated *type* to have primitive data type. A *derived* feature identifies which attributes are derived from other values. The simplified structural metamodel of relational schemas is shown on the right part of Fig. 1. *Tables* contain *Columns*, *Columns* can *refer* to other *Tables* in case of foreign keys.

In Listing 1.1 we use the CoqTL language to specify how to transform class diagrams to relational schemas. A transformation is a Coq *Definition*. First, we declare that a transformation named *Class2Relational* is to transform a model conforming to the *Class* metamodel to a model conforming to the *Relational* metamodel, and we name the input model as m (lines 2–3).

Then, the transformation is defined via two rules in a mapping style: one maps *Classes* to *Tables*, another one maps *non-derived Attributes* to *Columns*. Each rule in CoqTL has a *from* section that specifies the input pattern to be matched in the source model. A boolean expression in Gallina can be added as guard, and a rule is applicable only if the guard evaluates to true for a certain assignment of the input pattern elements. Each rule has a *to* section which specifies elements and links to be created in the target model (output pattern) when a rule is fired. The *to* section is formed by a list of labeled *outputs*, each one including an *element* and a list of *links* to create. The *element* section includes standard Gallina code to instantiate the new element specifying the value of its attributes (line 13). The *links* section contains standard Gallina code to instantiate links related to the previous element (lines 17–19).

For instance in the *Class2Table* rule, once a class *c* is matched (lines 6 to 8), we specify that a table should be constructed by the constructor *BuildTable*, with the same *id* and *name* of *c* (line 13). While the body of the *element* section (line 13) can contain any Gallina code, it is type-checked against the *element* signature (line 12), i.e. in this case it must return a *Table*.

In order to link the generated table *t* to the columns it contains, we get the *attributes* of the matched class (line 17), resolve them to their corresponding *Columns*, generated by any other rule (line 18), and construct new set of links connecting the table and these columns (line 19). While this is standard Gallina code, we use for this example an imperative style with a monadic notation (←, similar to the do-notation in Haskell) that makes the code more clear in this case[2]. The *resolveAll* function will only return the correctly resolved attributes. In particular derived *Attributes* do not generate *Columns* (i.e. they are not matched by *Attribute2Column*), so they will be automatically filtered out by *resolveAll*. The result of this Gallina code (i.e. the constructed links) are type-checked against the *link* signature (i.e. in this case they must have type *TableColumns*, as specified at line 16).

In the *Attribute2Column* rule we can notice the presence of a guard. When the *Attribute* is not *derived*, a *Column* is constructed with the same name and identifier of the *Attribute*. If the original attribute is *typed* by another *Class* we build a *reference* link to declare that the generated *Column* is a foreign key of a *Table* in the schema. This *Table* is found by resolving (*resolve* function) the *Class* type of the attribute.

CoqTL naturally enables deductive verification of model transformations. Users can write Coq theorems that apply pre/postconditions (correctness conditions) to the model transformation. For example, Listing 1.2 defines a theorem stating that if all elements contained by the input model have not-empty *names*, by executing the *Class2Relational* MT, all generated elements in the output model will also have not-empty *names*. Interactively proving this simple

[2] The intuitive semantics of ← is: if the right-hand-side of the arrow is not *None*, then assign it to the variable in the left-hand side and evaluate the next line, otherwise return *None*.

```
Theorem Table_name_definedness :
  ∀ (cm : ClassModel) (rm : RelationalModel),
    (* transformation *)
    rm = execute Class2Relational cm
    →
    (* precondition *)
    (∀ (i : Class), In i (allModelElements cm)→ length (getClassName i)>0)
    →
    (* postcondition *)
    (∀ (o : Table), In o (allModelElements rm)→ length (getTableName i)>0).
```

Listing 1.2. Name definedness theorem for the *Class2Relational* transformation

theorem in Coq takes 56 lines of routine proof code (this short proof can be even automated by using modern automatic theorem provers [3,6]).

To illustrate more complex theorems we want to prove that our transformation *preserves unreachability*. (Un)reachability is an important property for several models, e.g. one may typically need to demonstrate that error states in generated state machines are not reachable. In our simple Class2Relational example, one can inductively define reachability for classes (similarly for tables), i.e. a class is reachable from itself, and two classes are reachable if they are transitively linked by attributes. We can define an *unreachability preservation* theorem as follows: if a certain class is not reachable from a given class, their corresponding tables will not be reachable from each other. Interactively proving this theorem in CoqTL needs more than a thousand lines of proof code. The major difficulty comes from choosing the right induction strategy, and to our knowledge, the automatic proof of similar theorems is not addressed by existing work. The full proof in Coq is available on the paper website.

3 The Design of CoqTL

CoqTL is an internal DSL for model transformation in Coq. In this section we will describe the three main parts of its design:

- (Section 3.1) Metamodels and models are encoded as graph structures that can be automatically translated from/to EMF.
- (Section 3.2) Transformation specifications are encoded as a data structure wrapped up in a user-friendly domain specific syntax.
- (Section 3.3) A transformation engine interprets transformation specifications against input models.

3.1 Metamodels and Models

Our encoding of metamodels in Coq is similar to analogous encodings in related work, based on inductive data types. As an example, Listing 1.3 shows the basic definitions for encoding the *Relational* metamodel of Fig. 1. Since this interface is the main means to access source and target models, we aim at providing the simplest native representation.

Each metaclass is represented by an inductive data type, with a single constructor whose arguments are the attributes of the metaclass. References between metaclasses are represented as separate inductive types, with a constructor requiring the source and target elements as arguments. Optional or multivalued attributes and references are respectively represented using the *option* and *list* Coq types in the appropriate constructor argument (e.g. at line 15).

Constructing any model requires providing a list of model elements and one of links, as specified by the *Model* type in the CoqTL library (shown in lines 23–26 in the listing). These lists are typed by generic *ModelElement* and *ModelLink* types, that are meant to be the sum types for elements and links of the specific metamodel. For defining the type of Relational model, we first define the two sum types *RelationalModelElement*, sum of *Table* and *Column*, and *RelationalModelLink*, sum of *TableColumns* and *ColumnReference* (for simplicity here we omit the definition of sum types, that relies on dependent types). The *RelationalModel* type is obtained by parametrizing *Model* with these sum types.

We create accessors for every attribute and reference of each metaclass. Notice that while attribute accessors need only to inspect the element passed as argument to retrieve the attribute value (e.g., *getTableId* and *getTableName* at lines 37–44), reference accessors need to pass through the list of links to find the ones connected to the element in parameter. Thus, reference accessors need to have the whole model as extra parameter (e.g., *getTableColumns* in the listing).

Listing 1.3 shows a small portion of the encoding of the *Relational* metamodel in Fig. 1. The full encoding takes over 300 lines of Gallina code, and includes a reflective API. Briefly, metamodel classes are reified in a RelationalMetamodelClass type (with values corresponding to Table and Column), that is used as argument to reflective functions. The reflective API can be used for obtaining the metaclass of an element, checking that an element is an instance of a metaclass, and casting a generic element to/from a specific metaclass. Similar functions are available for links.

While our representation allows us to encode any model instance, in our current prototype we do not directly implement several features that are found in modeling frameworks like EMF. Bidirectional references currently have no special treatment: both sides are encoded as separate references, that need to be separately assigned in the transformation code. No direct support is provided for metaclass inheritance: the instance of a superclass can be provided as parameter of a subclass constructor, but the two instances (of superclass and subclass) need to be managed separately. Constraints for reference multiplicity or strong containment can only be encoded via extra pre/postconditions. Finally, differently from EMF, identifiers are considered as normal attributes and elements are considered equal when all their attributes are.

Automatic translators to/from EMF are still under development, and only partial implementations are provided on the CoqTL website.

```
(*** Metamodel classes and references ***)

Inductive Table : Set :=
    BuildTable :
      (* id *) nat →
      (* name *) string → Table.

Inductive Column : Set :=
    BuildColumn :
      (* id *) nat →
      (* name *) string → Column.

Inductive TableColumns : Set :=
    BuildTableColumns:
      Table → list Columns → TableColumns.

Inductive ColumnReference : Set :=
    BuildColumnReference:
      Column → Table → ColumnReference.

(*** Model (from CoqTL library) ***)

Inductive Model (ModelElement: Type) (ModelLink: Type): Type :=
    BuildModel:
        list ModelElement →
        list ModelLink → Model ModelElement ModelLink.

(*** Relational Model ***)

Inductive RelationalModelElement : Set := ...   (* sum type for elements *)
Inductive RelationalModelLink : Set := ...   (* sum type for links *)

Definition RelationalModel := Model RelationalModelElement RelationalModelLink.

(*** Table accessors ***)

Definition getTableId (t : Table) : nat :=
    match t with BuildTable id _ ⇒ id end.

Definition getTableName (t : Table) : string :=
    match t with BuildTable _ n ⇒ n end.

Definition getTableColumns (t : Table) (m : RelationalModel) :
    option (list Columns) := ...
```

Listing 1.3. Some basic definitions for the *Relational* models in Coq

3.2 Transformation Specification

Grammar 1.1 describes the concrete syntax of CoqTL. With respect to what we already discussed in Sect. 2, the grammar shows that CoqTL supports patterns with multiple input and output pattern elements. As indicated by the *header* production rule, CoqTL currently supports only transformations from a single source model to a single target model.

⟨*transformation*⟩ ::= ⟨*header*⟩ ‘`:=`’ ‘`[`’ ⟨*rule-list*⟩ ‘`]`’

⟨*header*⟩ ::= ‘`transformation`’ ‘`from`’ ⟨*id*⟩ ‘`to`’ ⟨*id*⟩ ‘`with`’ ⟨*id*⟩ ‘`as`’ ⟨*id*⟩

⟨*rule-list*⟩ ::= ⟨*rule*⟩ ‘`;`’ ⟨*rule-list*⟩ | ⟨*rule*⟩

⟨*rule*⟩ ::= ‘`rule`’ ⟨*id*⟩ ‘`from`’ ⟨*input-pattern*⟩ ‘`to`’ ⟨*output-pattern*⟩

⟨*input-pattern*⟩ ::= ⟨*elem-decl-list*⟩ ‘`when`’ ⟨*gallina-expr*⟩

⟨*elem-decl-list*⟩ ::= ⟨*elem-decl*⟩ ‘`,`’ ⟨*elem-decl-list*⟩ | ⟨*elem-decl*⟩

⟨*elem-decl*⟩ ::= ‘`element`’ ⟨*id*⟩ ‘`class`’ ⟨*id*⟩ ‘`from`’ ⟨*id*⟩

⟨*output-pattern*⟩ ::= ‘`[`’ ⟨*output-list*⟩ ‘`]`’

⟨*output-list*⟩ ::= ⟨*output-elem*⟩ ‘`;`’ ⟨*output-list*⟩ | ⟨*output-elem*⟩

⟨*output-elem*⟩ ::= ‘`output`’ ⟨*string*⟩ ‘`element`’ ⟨*elem-def*⟩ ‘`links`’ ‘`[`’ ⟨*link-def-list*⟩ ‘`]`’

⟨*elem-def*⟩ ::= ⟨*elem-decl*⟩ ‘`:=`’ ⟨*gallina-expr*⟩

⟨*link-def-list*⟩ ::= ⟨*link-def*⟩ ‘`;`’ ⟨*link-def-list*⟩ | ⟨*link-def*⟩

⟨*link-def*⟩ ::= ⟨*link-decl*⟩ ‘`:=`’ ⟨*gallina-expr*⟩

⟨*link-decl*⟩ ::= ‘`reference`’ ⟨*id*⟩ ‘`from`’ ⟨*id*⟩

Grammar 1.1. The concrete syntax of the CoqTL language

The way we implement the concrete syntax of CoqTL relies on the Notation facility of Coq. A notation is a symbolic abbreviation to denote some expressions, and is one of the main commands that modifies the way Coq parses and prints the representation of expressions.

```
(* Output Link Definition *)
Notation "'reference' reftype 'from' tinstance ':=' refends" :=
  (BuildOutputPatternLinkDefinition tinstance reftype refends)
    (right associativity, at level 60).

(* Output Pattern Element *)
Notation "'output' elid 'element' elname 'class' eltype
  'from' tinstance := eldef 'links' refdef" :=
  (BuildOutputPatternElement eltype elid eldef (fun elname ⇒ refdef))
    (right associativity, at level 60).
```

Listing 1.4. A few notations for CoqTL

For example, the first notation shown in Listing 1.4 implements the production rules *link-def* and *link-decl* in Grammar 1.1. After the declaration of this notation, when the expression on the left-hand-side is matched, it is expanded in memory to the right-hand-side. A notation allows also the specification of associativity and precedence levels, to solve parsing ambiguities. Notations can be seen as a very limited compiler, that compiles in one pass without memory. For this reason they strongly limit the classes of DSLs that can be implemented. In the implementation of CoqTL every notation is simply translated into an appropriate constructor, encapsulating the values matched by the notation (line 3). Whenever the notation is matching the declaration of some variable that needs to be visible to the rest of the code, we introduce a lambda expression as an argument of the constructor. This is shown in the second notation in Listing 1.4, that implements the *output-elem* production rule in the grammar. The created element *elname* needs to be visible in the following *links* section, so we store the content of this section in an anonymous function with *elname* as input (line 9).

The constructors used in our notations, like *BuildOutputPatternLinkDefinition* in Listing 1.4 build a representation of the abstract syntax of the CoqTL program. Hence CoqTL is a deeply embedded DSL for the rule structure part. CoqTL has however shallow embedding of expressions, to allow the direct use of the Gallina language for guards and output patterns (*gallina-expr* in the grammar).

Gallina has several characteristics that make it suitable as an expression language for CoqTL. It is:

– Expressive. Gallina is based on a formal language called the Calculus of Inductive Constructions, combining a higher-order logic and a richly-typed functional programming language.
– Easy to learn. In our experience, the learning curve of the language is low if the user had some exposure to functional languages.
– Accompanied by sophisticated libraries. Reusing functions in those libraries during the MT specification is also important for the proof phase, that can exploit the theorems and lemmas provided by the library for those functions.

Finally, CoqTL provides auxiliary functions meant to be used in Gallina expressions for guards and output patterns. The most important is the function *resolve* (and its corresponding multivalued version, *resolveAll*) for element resolution. As illustrated at lines 18 and 36 in Listing 1.1, its signature requires the following arguments: (1) the result of the matching phase of the current transformation (*match Class2Relational m*), (2) the label associated to the required output element, useful for rules with multiple output elements ("*col*"), (3) the type of the expected result, useful for type checking (*Column*), (4) the source pattern to resolve (or the list of source patterns in case of *resolveAll*). Notice that the matching phase is provided as a new application of the transformation in a specific *match* mode. While this choice affects the global efficiency of the transformation, it simplifies the development of proofs, because it avoids having a concept of transformation traces as side effects of the transformation execution.

While the expressiveness of CoqTL has important limitations, we are currently able to manually translate to CoqTL a significant subset of ATL transformations: 1-to-1 model transformations, in standard (non-refining) mode, written in the declarative subset of ATL, without lazy rules.

3.3 Transformation Engine

Algorithm 1 illustrates in pseudocode how the transformation specifications are interpreted by our transformation engine. This algorithm has been influenced by the execution algorithm of ATL [12] (notably in the distinction between a match/instantiate and an apply function), but is very different, having the objective to simplify the proof development, at the cost of sacrificing execution efficiency.

Our transformation engine is implemented in an *execute* function (called for instance in Listing 1.2) that takes as input a transformation specification R and an input model I (which contains elements I_e and links I_l). The output is elements O_e and links O_l, which form an output model.

First, the transformation engine records the maximum size (m) of input patterns among all the rules in the transformation specification. This value is used to calculate all the potential pattern instances P that the input model can produce to be matched against the transformation specification, i.e. all the subsets of I_e whose size is less or equal to m are enumerated.

Next, the engine iterates on each potential pattern instance p, and seeks for a rule r in R that matches it (i.e. if model elements in the pattern instance have the types defined in the input pattern of the rule) and satisfies the guard of that rule. If a rule r is found for the pattern instance p, then the *instantiation* phase of r will be invoked to construct the corresponding output elements of p and add them to the output model. Finally the *apply* phase is invoked, i.e. to construct the corresponding output links and add them to the output model.

Algorithm 1. Algorithm of the *execute* function

```
1: m ← maxArity(R)
2: P ← allPatterns($I_e$, m)
3: for each $p$ ∈ P do
4:     r ← findRule(R, p)
5:     if $r \neq$ None then
6:         $O_e \leftarrow O_e \cup$ instantiate(r, p)
7:         $O_l \leftarrow O_l \cup$ apply(r, p)
8:     end if
9: end for
```

Notice that Gallina expressions for output links are only evaluated during the apply phase. The developer may include in these expressions calls to the *resolve* or *resolveAll* functions, whose evaluation requires the execution of the *instantiate* phase. As mentioned in the previous section, in our solution the user passes to *resolve* the result of the transformation execution in *match* mode (i.e. *match* function at (lines 18 and 36 in Listing 1.1). The algorithm implemented in *match* is identical to Algorithm 1, without line 7 (that would make the whole computation recur indefinitely). Multiple executions of the transformation for element resolution slow down the execution, but simplify the proofs, since no explicit traces are necessary as applications of *instantiate* and *apply* with identical inputs can be trivially checked for equality. Possible optimizations are however the subject of future work.

Finally the application of the transformation by the *execute* function can be automatically *extracted* by Coq into a separate executable program in several languages (e.g., OCaml, Haskell).

4 Proving Theorems with CoqTL

In this section we show that CoqTL can enable practical verification for MTs. We formulate 4 theorem proofs over the model transformation presented in Sect. 2. Some measures are shown in Table 1, to give the reader an idea of the complexity of the proofs: lines of code (LoC) and number of user-developed lemmas.

Table 1. Theorem proofs on *Class2Relational*

Theorem	LoC	No. Lemmas
positive_ids	180	4
positive_ids_surj	75	1
name_definedness	89	2
Unreachability_preservation	1161	17

As a first theorem we prove that Class2Relational preserves id positivity, i.e. if all identifiers in the source model are positive, then they also are in the target model. In the first and second row we show two proofs for this theorem. In the second proof we obtain a reduction of about 60% LoC, thanks to the use of a generic lemma for transformation surjectivity, provided in the CoqTL library. This shows that CoqTL enables the design and proof of generic theorems that make interactive verification more efficient and concise.

Transformation surjectivity states that for all elements contained in the output model there has to exist a rule and a matching input pattern that created them. Our design choices in CoqTL enable this kind of theorems: during the proof we can refer to syntactic elements of the transformation (e.g. *rules*, *input/output patterns*) by their type in the abstract syntax (e.g., OutputPatternLinkDefinition in Listing 1.4), and *quantify over them*. Moreover we use the reflective model API mentioned in Sect. 3.1 to reason on metamodel-agnostic properties.

The surjectivity lemma is also used in the third and fourth proof. In the third row we prove the name definedness property shown in Listing 1.2, separately for all element types in source and target models. Finally by the fourth row, it is clear that the unreachability preservation theorem (Sect. 2) is difficult to prove, and shows the need of further work in proof engineering for MTs.

One road we want to follow is providing a complete library of generic lemmas for CoqTL such as transformation surjectivity, to shorten proofs on CoqTL. Some recurring proof patterns could be factorized into *domain-specific automatic proof tactics*, aware of the CoqTL representation and properties. Another line could be investigating a set of domain specific guidelines to construct proofs for MT verification. For example, to prove that if two *Tables* are reachable, the *Classes* that generated them are reachable too; we induct on the definition of reachability. However other induction strategies, e.g. on the structure of the model, may be more efficient.

5 Related Work

There are many automatic theorem proving approaches for MTs (e.g. [3,4,6,16]). However, interactive theorem proving is inevitable for more serious verification tasks. In this section, we focus on recent advancements of MT verification based on interactive theorem proving. To our knowledge, none of the existing works designs and implements DSLs for MT within interactive theorem provers.

Yang et al. interactively verify that a particular model transformation, i.e. from AADL to TASM language, is semantic preserving [23]. The approach is based on providing a translational semantics of both languages as timed transition systems in Coq and then reasoning on their equivalence. CoqTL could be used to simplify this kind of work.

Most previous works focus on giving a translational semantics of a MT language towards the target theorem prover. Generally they do not investigate a way to formally ensure that the semantics of the MT language has been axiomatized correctly in the back-end theorem prover. Calegari et al. encode ATL MTs and OCL contracts into Coq to interactively verify that the MT is able to produce target models that satisfy the given contracts [5]. In [21], a Hoare-style calculus is developed by Stenzel et al. in the KIV prover to analyze transformations expressed in (a subset of) QVT Operational. UML-RSDS is a tool-set for developing correct MTs by construction [14]. It chooses well-accepted concepts in MDE to make their approach more accessible by developers to specify MTs. Then, the MTs are verified against contracts by translating both into interactive theorem provers.

Kezadri et al. defines the Coq4MDE framework to formally embed some key aspects of MDE in Coq [11]. We have a similar abstraction of metamodels as graphs. While our understanding is that Coq4MDE is capable of embedding MT languages and enabling MT verification, no specific work has been proposed. We expect an evaluation in the future to compare the complexity of MT verification between the two works.

Poernomo and Terrell follow the classical approach in type theory to formally specify MTs as ∀∃ types in interactive theorem provers [19]. Their approach does not target any specific MT languages. In addition, although their work does not propose a generic MT engine as we presented here, a corresponding executable MT program can be extracted once the MT is proved. The approach is further extended by Fernández and Terrell on using co-inductive types to encode bi-directional or circular references [9]. We also plan to investigate how co-inductive types can cooperate with our encoding and proofs (e.g. guardedness issues of co-recursive functions might arise because the syntactic criterion applied by the Coq system is too rigid [17]).

6 Conclusion

In conclusion, we present CoqTL, to our knowledge the first DSL in Coq for MTs and their verification. CoqTL is both functional and declarative in style, providing a familiar environment for transformation developers in Coq. Its underlining transformation engine, implemented in Coq, allows CoqTL programs to be interpreted against input models to compute output models. We show the practical applicability of CoqTL, by proving non-trivial contracts over a sample transformation.

Our future work would focus on the issues we identified in different points of our discussion. We want to develop a theorem library on top of CoqTL to facilitate MT verification, including of transformation-agnostic lemmas such as transformation surjectivity and domain-specific proof tactics to automatize recurring proof steps. We aim to investigate whether there are domain-specific guidelines to construct proofs for MT verification. We want to improve interoperability between CoqTL and common MDE tools such as EMF, for industry readiness.

Acknowledgements. We thank Rémi Douence for his valuable help during the development of CoqTL.

References

1. Rahim, L.Ab., Whittle, J.: A survey of approaches for verifying model transformations. Softw. Syst. Model. **14**(2), 1003–1028 (2015)
2. Berry, G.: Synchronous design and verification of critical embedded systems using SCADE and esterel. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, p. 2. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79707-4_2
3. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 432–448. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_28
4. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 198–213. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_16
5. Calegari, D., Luna, C., Szasz, N., Tasistro, Á.: A type-theoretic framework for certified model transformations. In: Davies, J., Silva, L., Simao, A. (eds.) SBMF 2010. LNCS, vol. 6527, pp. 112–127. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19829-8_8
6. Cheng, Z., Monahan, R., Power, J.F.: A sound execution semantics for atl via translation validation. In: Kolovos, D., Wimmer, M. (eds.) ICMT 2015. LNCS, vol. 9152, pp. 133–148. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21155-8_11
7. Chlipala, A.: The Bedrock structured programming system: combining generative meta programming and hoare logic in an extensible program verifier. In: 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, pp. 391–402. ACM, Boston (2013)

8. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: RubyTL: a practical, extensible transformation language. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 158–172. Springer, Heidelberg (2006). https://doi.org/10.1007/11787044_13
9. Fernández, M., Terrell, J.: Assembling the proofs of ordered model transformations. In: 10th International Workshop on Formal Engineering approaches to Software Components and Architectures, pp. 63–77. EPTCS, Rome, Italy (2013)
10. Gu, R., Shao, Z., Chen, H., Wu, X., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: an extensible architecture for building certified concurrent OS kernels. In: 12th USENIX Conference on Operating Systems Design and Implementation, pp. 653–669. USENIX Association, Berkeley (2016)
11. Hamiaz, M.K., Pantel, M., Combemale, B., Thirioux, X.: A formal framework to prove the correctness of model driven engineering composition operators. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 235–250. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11737-9_16
12. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
13. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69927-9_4
14. Lano, K., Clark, T., Kolahdouz-Rahimi, S.: A framework for model transformation verification. Formal Aspects Comput. **27**(1), 193–235 (2014)
15. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. SIGPLAN Not. **41**(1), 42–54 (2006)
16. Oakes, B.J., Troya, J., Lúcio, L., Wimmer, M.: Fully verifying transformation contracts for declarative ATL. In: 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 256–265. IEEE, Ottawa (2015)
17. Picard, C., Matthes, R.: Coinductive graph representation: the problem of embedded lists. Electron. Commun. EASST **39** (2011)
18. Pierce, B.C., de Amorim, A.A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Software Foundations. In: Electronic Textbook (2017)
19. Poernomo, I., Terrell, J.: Correct-by-construction model transformations from partially ordered specifications in Coq. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 56–73. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16901-4_6
20. Selim, G.M.K., Wang, S., Cordy, J.R., Dingel, J.: Model transformations for migrating legacy models: an industrial case study. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 90–101. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_9
21. Stenzel, K., Moebius, N., Reif, W.: Formal verification of QVT transformations for code generation. Softw. Syst. Model. **14**, 981–1002 (2015)
22. Wagelaar, D.: Using ATL/EMFTVM for import/export of medical data. In: 2nd Software Development Automation Conference, Amsterdam, Netherlands (2014)
23. Yang, Z., Hu, K., Ma, D., Bodeveix, J.P., Pi, L., Talpin, J.P.: From AADL to timed abstract state machines: a verified model transformation. J. Syst. Softw. **93**, 42–68 (2014)

A Formal Framework for Prototyping Executable Semantics in ATL

Artur Boronat(✉)

Department of Informatics, University of Leicester, Leicester, UK
aboronat@le.ac.uk
http://arturboronat.info

Abstract. ATL is a well-established model transformation language both in industry and in academia, where it is used as a reference language for studying different types of model transformations and their properties. In this paper, we discuss current limitations of ATL's in-place semantics that hamper its application for modelling and verifying systems and propose a new in-place semantics for ATL that enables it as a specification language for simulating and verifying EMF-based systems. Our approach is based on FMA-ATL, an executable specification of a large excerpt of ATL in Maude, which has been augmented with the new in-place semantics so that Maude's verification tools can then be used both to perform bounded model checking of invariants and to model check LTL formulas in the resulting system models, where appropriate. Furthermore, FMA-ATL uses ATL as front-end language and it can be *reused as-is* for verification, including its tool support.

Keywords: EMF · ATL · System specification · Formal methods

1 Introduction

ATL is a model-to-model transformation language that seeks pragmatism by ensuring that executed model transformations always produce the same result. This offloads the responsibility of ensuring those conditions from software engineers when designing a model transformation, which helps to focus on the domain problem, namely the transformation definition. Such pragmatism is implemented by using a read-only source model in which model elements are transformed only once, building an output model from scratch. There are situations where such bulk semantics is too expensive, both from a productivity point of view and from a computational point of view. For example, when endogenous model transformations involve sparse model updates in possibly large models, explicit rules need to be introduced in order to copy the model elements that are not the target of model updates and that are to be preserved.

The ATL2010 compiler addresses these concerns by emulating in-place model transformations [10] for ATL transformations in refining mode using a two-step process, which relies on an abstract language for defining updates. In a first step, rules are applied and a diff model is computed representing in-place changes as a patch of model differences. In a second step, the changes obtained from all the rules are reordered in order to apply creations first, modifications afterwards and deletions at the end, ensuring the standard properties of ATL transformations. The resulting sequence of model changes is applied as a patch to the model.

One could consider the use of in-place ATL transformations for specifying and simulating software systems by modelling system states with EMF (Eclipse Modeling Framework) models and by capturing the dynamic aspects of the system with an ATL modules. System simulation could be achieved by successive applications of the transformation to pre-states in order to produce post-states, with the amalgamated updates of several rule applications. However, ATL in-place semantics presents a number of drawbacks that hampers such an approach. By the very nature of ATL, transformations are deterministic when they are evaluated by ensuring that each source object is matched by an ATL rule only once. This means that only a subset of deterministic systems can be modelled for verification purposes. Non-deterministic systems and, hence, concurrent systems cannot be modelled with ATL transformations. In particular, there are two reasons that hinder *soundness* and *completeness* of the verification of systems modelled with ATL. First, when a transformation is applied to a source model, ATL's in-place strategy affects side-effects but not the matches of rules, which are computed up front. For example, the application of a rule that disables a pre-computed match is disregarded, yielding an incorrect result. Second, the application of rules that enable new matches are also disregarded for the same reason. This means that in-place ATL transformations may not capture all the intended behaviour of a system. That is, an in-place model transformation model, understood as the class of model transformations for all models conforming to the source metamodel, cannot be used as a system model, corresponding to execution paths between system states. This is important from a verification point of view, as the absence of errors in that case is no guarantee of the correctness of the actual system. That is, the specification is an under-approximation of the intended behaviour. These drawbacks are illustrated with the running example of Sect. 2 in 5.

In this work, the structural operational semantics (SOS) of FMA-ATL [2], which formalizes a large excerpt of representable ATL model transformations, has been augmented with a new in-place semantics that overcomes those problems. This semantics specification is implemented faithfully in Maude [3] yielding a scheduler for applying matched rules and an interpreter for their side effects. FMA-ATL uses the EMF as front-end for defining metamodels and models, permitting the reuse of EMF-based systems and domain-specific modelling languages (DSMLs). Furthermore, our approach reuses Maude's verification tools for analysing correctness properties in the resulting system models. Furthermore, we use the official ATL language as the front-end language for FMA-ATL, providing a new engine for ATL equipped with formal verification techniques. This last contribution facilitates the validation and verification of ATL system specifications by reusing the tool ecosystem that is already available for ATL, facilitating the collaboration between software engineers specialized in (model-driven) software development and software engineers specialized in validation and verification. The tool and examples used are available at https://fma-atl.github.io/.

In the rest of the paper: in Sect. 2, ATL is presented as a specification language for EMF-based systems, using the *Concurrent Append Problem* from [9] as a running example; in Sect. 3, ATL is used as property specification language and the different verification techniques supported in FMA-ATL are illustrated; in Sect. 4, the integration of ATL is discussed; in Sect. 5, the shortcomings of ATL2010's in-place semantics are illustrated with the running example and FMA-ATL features are compared against those of representative tools used for simulating and verifying EMF-based systems; and final concluding remarks are given in Sect. 6.

2 ATL as System Specification Language

Combining refining mode and in-place transformations without control flow constraints removes the guarantees that make ATL transformations confluent and terminating. Fortunately, these are also the conditions that enable ATL as a specification language for modelling concurrent processes, which we consider in the rest of this section. We first introduce an example that cannot be modelled using the in-place semantics of ATL, explaining the syntax used to model EMF-based systems, and then show how this semantics is captured by augmenting the FMA-ATL semantics with a new scheduler rule.

2.1 The Concurrent Append Problem

Our running example is the *concurrent append problem* of the Java program of Fig. 1, adapted from [9], which implements the append method on cells, which may be arranged forming a list, of Fig. 1. Given a String value `x` as parameter, the program appends a new tail cell to the list if `x` is not contained in any of the existing cells. An example correctness criterion is that the list of cells must not contain the same value more than once. However, different threads may access cells concurrently by calling the append method, which might result in undesired race conditions without certain assumptions on atomicity.

In this paper, the example is modelled using the ATL system specification of Fig. 2, where the state model, represented as an EMF model, and an initial state are given in Fig. 1. The state model expresses that each `cell` contains a value `val` and may have a successor via the composition `next`. The append method call is represented with: an `Append` object with the argument `x` to be inserted; an `active` flag (corresponding to the program counter) indicating that the method is being executed; and a `return` flag indicating that the execution of the method is over. Recursion is modelled using the containment reference `callee`.

States are represented as nested object diagrams, where objects may be nested through containment references. Each containment reference is depicted as a labelled box, whose contents are the immediate children. The state of Fig. 1 shows two concurrent calls to `append("b")` on the singleton list `["a"]`.

```
class Cell {
  Cell next;
  int val;
  void append(int x) {
    if (x == this.val)
      return;
    if (this.next == null) {
      this.next = new Cell();
      this.next.val = x;
    } else
      this.next.append(x);
  }
}
```

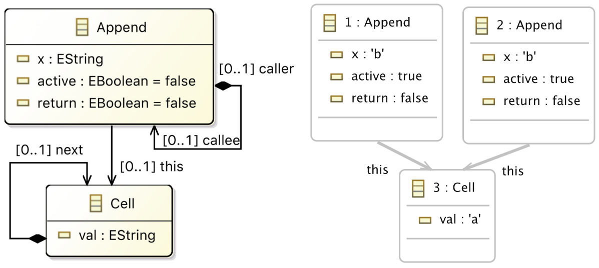


Fig. 1. Java program (left), model of states (middle) and initial state (right).

In the following, we explain how to use ATL to model the `Append` method using a new in-place semantics, an defer the discussion of deviations w.r.t the current in-place semantics in Subsect. 5.1.

In ATL, a (matched) specification rule has: a name; a source pattern, denoted with the keyword `from`, consisting of an object variable (or element) to be matched for the rule to be enabled and a filter condition expressed in OCL where the object variable can be used; a potentially empty list of local variable initializations, enclosed by the `using` block; and a target pattern, denoted by the keyword `to`, containing a list of object variables (or elements) that refer to objects that are created, when a new variable is used, or to objects that are updated, when the name of a declared variable (either the source pattern variable or a local variable) is suffixed with `__ref`. Each target object variable encloses a list of bindings, each of which corresponds to a feature (attribute or reference) initialization, when the object is created, or to a feature update, when the `__ref` naming convention is used in the object variable name. Updates for attributes reset their value. Updates for references (including containments) either reset their value, if the upper bound is one, or append a new reference if the upper bound is not met. Updates may also be destructive for references if the suffix `__unset` is appended to the name of the reference. In that case, the corresponding reference is deleted if the lower bound is not met. A containment reference can only be deleted when the contained object is *isolated*, there are no incident references to it or to any of its contents. The deletion of a containment reference implements *cascade delete* semantics, that is the contained object and its contents vanish outright. ATL also provides the statement `drop` that can be used in the target pattern of a specification rule, denoting the deletion of the object matched by the source pattern, with the same *delete cascade* semantics.

An ATL specification can also contain lazy specification rules, which are not matched by the scheduler and have to be called from matched rules explicitly. In FMA-ATL [2], unique lazy rules are used to reduce the state space of a system specification from an initial state by amalgamating the side effects of all the lazy rule applications in one big transition step. Moreover, an ATL specification can contain helpers, which are functional operations that can be used to query the source model or to perform computations. ATL helpers will be discussed in Sect. 3.

The ATL specification modelling the dynamic behaviour of the `append` method, adapted from [9], is shown in Fig. 2, and consists of the following rules:

Append a new cell. Rule `Append` is responsible for appending a new cell to the list if the control reaches the last cell (there is an active `Append` object pointing to the last cell) and the value stored at this last cell is not equal to the method argument.

Go to next cell. Rule `Next` checks whether the method argument is not equal to the value stored at the current (`this`) cell and makes a recursive call then for checking the next cell by generating a new `Append` object and declaring it as the `active` call, deactivating the current call.

Value found in list. Rule `Found` checks if the method argument matches the value stored at the current (`this`) cell and, if so, indicates that the computation is over by disabling `active` and by enabling `return`.

Return result. Finally, rule `Return` simply removes an append invocation object (from the stack of recursive calls) if it has already calculated the result.

```
rule Append {
  from a1 : append!Append (
    a1.active and a1.this.val <> a1.x
    and a1.this.next.oclIsUndefined()
  ) using {
    c1 : append!Cell = a1.this;
  } to a1__ref : append!Append (
    x <- '',
    active <- false,
    return <- true
  ),
  c2 : append!Cell (
    val <- a1.x
  ),
  c1__ref : append!Cell (
    next <- c2
  )
}
rule Found {
  from a1 : append!Append (
    a1.active and a1.x = a1.this.val
  ) to a1__ref : append!Append (
    x <- '',
    active <- false,
    return <- true
  )
}
```

```
rule Next {
  from a1 : append!Append (
    a1.active=true and a1.x <> a1.this.val
    and a1.callee.oclIsUndefined()
  ) using {
    c : append!Cell = a1.this.next;
  } to a1__ref : append!Append (
    active <- false,
    x <- '',
    callee <- a2
  ),
  a2 : append!Append (
    active <- true,
    x <- a1.x,
    this <- c
  )
}
rule Return {
  from a1 : append!Append (
    a1.return = true and
    not(a1.caller.oclIsUndefined())
    and a1.callee.oclIsUndefined()
  ) using {
    caller : append!Append = a1.caller;
  } to caller__ref : append!Append (
    return <- true,
    callee__unset <- a1
  )
}
```

Fig. 2. An ATL version of the method `Cell::append(x: String)`.

The FMA-ATL engine consists of a scheduler rule that is applied to engine configurations, i.e. *states* of the FMA-ATL engine. Given an ATL system specification, the FMA-ATL engine parses matched/lazy rules and helpers, producing the initial engine configuration. This includes the computation of attribute helpers, caching their result. It then computes all enabling matches for matched rules, by considering their source pattern element and its filter condition. Then the scheduler starts the system simulation by selecting one enabling match and the corresponding ATL matched rule. The execution of a matched rule involves the interpretation of both a FMA statement representing the side effects in the system state. After these side effects are applied, continues the execution with the next enabling match until no more rules can be applied. In subsequent sections, we describe the engine configurations and how the scheduler rule is used to simulate ATL system specifications from an initial system state.

2.2 FMA-ATL Configurations and Engine Initialization

The main configuration types of the FMA-ATL engine are depicted in Fig. 3. The class `AtlMatchingConfig` represents the configuration of the engine for applying matched rules: a `ruleStore` and a `helperStore` with the set of ATL rules and the set of helpers, respectively, that are defined in the transformation; a `queryDomain` pointing to the domain that contains the source model and a set of `domains` that correspond to the different target models that are created by the transformation. A domain contains a `name` that identifies the domain, a `model` referring to a collection of objects, a `loc` map with locations for the objects in the model, and a factory `new` for obtaining fresh identifiers when a new object is created.

To simulate a system specification, the FMA-ATL engine first initializes an `AtlMatchingConfig` configuration, loading each specification rule into a rule store and helpers into a helper store. A FMA-ATL rule is initialized, by generating a FMA statement that models the side-effects on the state that are represented in the bindings of the target pattern of a specification rule, as described in [2].

A FMA statement can be regarded as a sequence of typical updates that can be performed in an EMF model instance. This initialization is performed by extracting a graph of side effects from the list of bindings of each target pattern element. Nodes are target pattern object variables and expressions representing a query (used in the initialization of the binding). Named edges are defined from object variables to expressions or to other object variables representing each binding. Once the graph is generated, FMA-ATL walks through the graph twice starting from the root object and following containment edges: first, it obtains a FMA statement that creates a tree of objects that initializes their containment references; second, for each object created in the first traversal, it initializes their attributes and non-containment references.

For the in-place semantics, the type graph of the graph that is used to represent the side effects of an ATL specification rule has been augmented with *update* and *drop* nodes. On the one hand, update nodes are obtained when the name of an object variable in the target pattern of a rule contains the suffix `__ref`

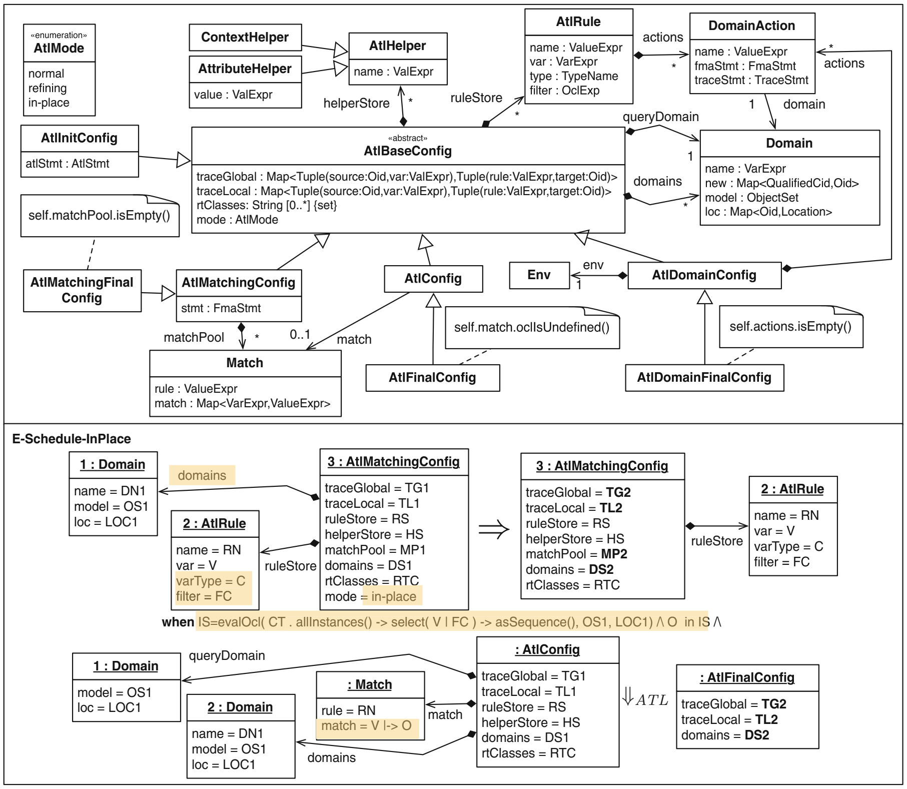


Fig. 3. FMA-ATL configuration model (top) and in-place scheduler rule (bottom).

and it coincides with the object variable used in a source object variable (either the source pattern object variable or a local variable of the `using` block). On the other hand, drop nodes are obtained when a `drop` statement is parsed. When a side-effect graph is translated into a FMA procedure, update nodes correspond to free variables that are to be bound in the environment. That is, FMA-ATL does not create a new object for update nodes in the first traversal of the graph. Moreover, `drop` nodes correspond to object destruction by deleting the corresponding containment reference when the object is not a root one.

Moreover, binding compilation to FMA statements has also been augmented by allowing deletion of references. This is used in an ATL specification by appending the suffix `__unset` to a reference name in a binding of a target pattern element. Such bindings are compiled to `unset` model actions in FMA when the graph of side-effects is traversed. In FMA, an `unset` action deletes a reference if the lower bound of the reference is not met. The deletion of a containment can only be applied when the object to be deleted or any of its contents are not referenced from an external object. Such a deletion entails the deletion of the objects, including its contents.

2.3 Rule Scheduling

In-place semantics for ATL specifications is defined by using a new scheduler rule `E-Schedule-InPlace`, shown in Fig. 3, where the main differences w.r.t. the normal (out-place) scheduler rule of [2] are highlighted. This rule is introduced in order to compute matches during the execution of the transformation, avoiding the up-front computation of the matches.

A match is computed as in the computation of matches up-front when ATL is executed in normal or refining mode. Given the contextual type `C` of the variable `V` used in the in pattern element of the rule and the filter condition `FC`, the list of matches for a rule is computed by evaluating the expression

```
CT.allInstances()->select( V | FC )->asSequence()
```

over the model `OS1` with the location map `LOC1` using the operation `evalOcl`. Thereby, the scheduler considers causal dependencies between rules based on the current state. In the implementation, rules are ordered lexicographically by name and the list of matches is ordered by each object internal id, a natural number. Therefore, each rule is applied for each list of matched objects orderly in the expression `O in IS` given that the list `IS` is computed for each rule application. This is, however, a potential source of *starvation* that needs to be taken into account when specifying a system: if a rule is always enabled for a list of objects, it will always be applied to the first object, treating others unfairly.

Once a match is found for a given specification rule, the match is defined by using the variable `V` of the in pattern element, and the side-effects of the specification rule, represented as a FMA statement, are interpreted using the big-step evaluation relation $\Downarrow_{ATL}$ presented in [2]. We can regard this evaluation relation as a black-box component where the precondition involves that a match for a rule must be selected and that the system state must be in the query domain (used to evaluate OCL queries) and in the domain (used to apply rule side effects). The postcondition of the evaluation relation guarantees that the system state in the resulting domain `DS2` is well-formed after applying the rule.

The new scheduler rule of Fig. 3 allows FMA-ATL to simulate system specifications. More specifically, this rule has been faithfully been implemented in Maude as a rewrite rule and each application of the `in-place` scheduler rule coincides with one system transition, thus executing the FMA-ATL engine from an initial configuration amounts to simulating the system specification from an initial system state. Furthermore, the `in-place` scheduler rule allows to reuse Maude's toolkit to traverse the system state space for verification purposes, as explained in the following section.

The behaviour of the system specification of our running example can thus be simulated by running the FMA-ATL engine with a system specification from an initial system state. Taking the system state of Fig. 1 as initial, a valid resulting execution path is graphically depicted in Fig. 4, denoting a rule application with an arrow, whose label contains the name of the arrow between system states, but for the initial state, and the identifier of the matched object (in between parenthesis).

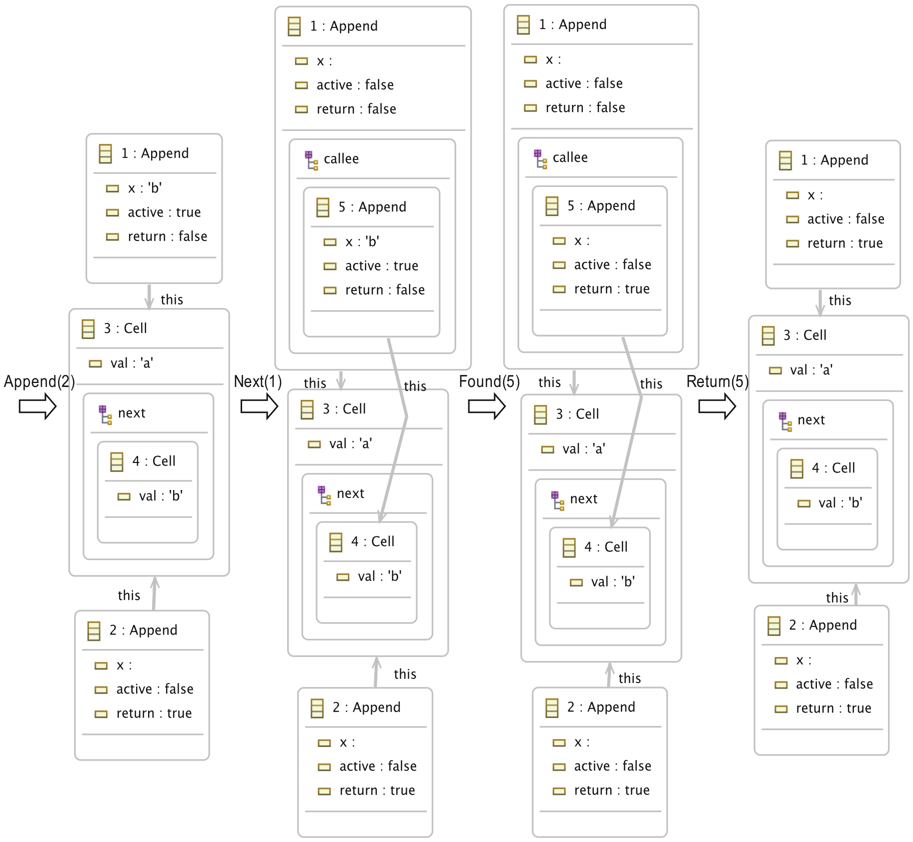


Fig. 4. Simulation from the initial system state of Fig. 1.

3 ATL as Property Specification Language

FMA-ATL is implemented in Maude so that we can reuse its LTL model checker for verifying temporal properties when the system specification models a finite state space [6], and its bounded model checker for invariants when the state space is infinite. The *model checking problem* consists in deciding whether a given correctness property holds in a specified system by systematically traversing all enabled transitions in all system states. That is, in FMA-ATL, given a system state, represented as an Ecore model instance, all possible enabled specification rules are applied and this procedure is recursively repeated on the successor states until no more matches are found.

Relevant classes of such correctness properties are *safety* and *reachability* properties. A safety property defines a desired property that should always hold on every execution path or (equivalently) an undesired situation which should never hold on any execution path. A reachability property describes, on the contrary, a desired situation which should be reached along at least one execution path. These two types of properties are interrelated in that a proof of the violation of a safety property is a witness of the reachability property defined as the negation of the safety property. Hence, if a safety property holds (or a reachability property is violated), the entire state space needs to be examined.

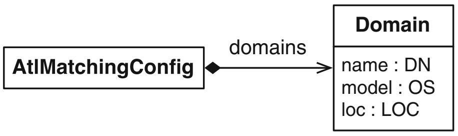
Such correctness properties are frequently formalized as LTL formulae built over a set of *state properties*, which either hold or not in a given system state. In FMA-ATL, the property specification is given in a separate ATL module defining the satisfaction of each state proposition using an ATL attribute of the form `helper def : P : Boolean = B ;` where `P` is the name of the state property and `B` is the boolean OCL expression that defines its satisfaction. The property specification module must share the same header with the system specification module. For example, in the listing below, the `Shared` property denotes when two distinct cells of the list contain the same value, a situation that is prohibited. Moreover, the `Isolated` property denotes a desired behaviour, a cell in the list and an append call will never share the same value in the same system state.

```
helper def: Shared : Boolean =
append!Cell.allInstances()->exists(c1 |
  append!Cell.allInstances()->exists(c2 | c1<>c2 and c1.val=c2.val )
);

helper def: Isolated : Boolean =
append!Cell.allInstances()->forAll(c |
  append!Append.allInstances()->forAll(a | a.x<>c.val )
);
```

To use Maude's model checker, the following components need to be characterized: the type of *states*, by defining a subsort of the sort `State`; the set of *state predicates* to be used as invariants or as atomic propositions in LTL formulas, by declaring them as subsort of the sort `Prop`; and finally the *satisfaction* of such state predicates, by providing equations for the operation

```
op _|=_ : State Prop -> Bool
```

In FMA-ATL, the set of states is defined by the class `AtlMatchingConfig` of our interpreter in Fig. 3. In that way, system states included in domains are wrapped by additional constructs that are used to specify the operational semantics. FMA-ATL declares a state property for each of the helper attributes defined in an ATL property specification module and defines its satisfaction using equations of the form[1]



$\models$ `P = evalOcl(B, OS, LOC)`

[1] Internally FMA-ATL works with a term representation of engine configurations and system states, which is depicted graphically for the sake of presentation.

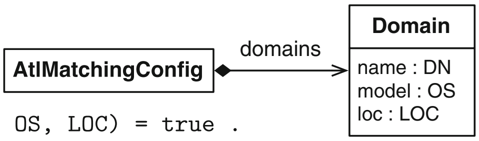
Then we can verify that the system satisfies the property that all cells in a given list will always contain unique values after a set of `append` calls, which may or may not contain the same values, with the following command:

```
red modelCheck( initialConfig , []~Shared ) .
```

where `~` denotes *not*, and `[]` is the LTL operator *always* ($\Box$) meaning that the property must hold in all future states, and `initialConfig` is the term resulting from the engine initialization phase as explained in Sect. 2.2.

Given an ATL system specification and property specification modules, an initial system state and the name `P` of an state property with body expression `B`, in the property specification module, FMA-ATL can verify invariants, such as `Isolated`, by traversing the state space using a breadth-first strategy with Maude's search command[2]

```
search initialConfig =>*
such that evalOcl(not B, OS, LOC) = true .
```



That is, the command searches for a configuration containing a system state where the expression `B` is violated. If such configuration is not found, the refutation process ends unsuccessfully and the invariant is satisfied because the state space is finite, in the example. For systems where the state space is infinite, an upper bound can be used for the analysis trading completeness for decidability.

4 Integration with ATL

FMA-ATL is available[3] as an EMF-based standalone library that can be used to execute a substantial excerpt of ATL model transformations. It enables formal verification of systems where the specification language, both for systems and for state properties, is ATL itself.

The execution of out-place model transformations, which was presented in [2], has been augmented with new functionality developed for this work: *(1)* integration with the ATL language; *(2)* simulation of model-based systems using ATL as specification language (with in-place matched rules); *(3)* bounded model checking of invariants, which are specified in ATL, when the state space of the specified system is infinite; and *(4)* software verification using LTL model checking, where state properties are specified in ATL, when the state space of the specified system is finite.

The front-end language for defining metamodels and system state models is EMF (Ecore) and the language for specifying model transformations, system

[2] Using `=>*` the search will be performed along zero or many simulation steps. However, other strategies that can be used are `=>!` for run to completion semantics, `=>1` for one step, `=>+` for at least one step.

[3] https://fma-atl.github.io.

specifications and property specifications is ATL. To implement the integration with ATL, FMA-ATL reuses parts of AnATLyzer [4] to infer types from ATL expressions and extends its ATL serializer to serialize ATL transformations to FMA-ATL. In FMA-ATL, expressions that are specific to ATL and extraneous to OCL, like `resolveTemp`, are evaluated independently of OCL expressions so that a Maude implementation of OCL, `mOdCL` [8], can be reused. This means that ATL expressions have to be transformed in order to extract ATL specific expressions (`resolveTemp` expressions, invocation of helpers and attributes, invocation of lazy rules) from OCL expressions, which requires transforming local variables (iterator variables) into global variables (FMA variables) while using unique names and remembering the scope where they are used.

5 Related Work

In this section, we analyse our contribution w.r.t. related work by looking at the differences with ATL in-place semantics in detail and, then, by providing a broader view of the features of FMA-ATL.

5.1 Differences with ATL In-Place Semantics

From a system specification point of view, when using ATL2010 in refining mode, transformation rules can match several objects by means of the `using` block, and several objects can be added to the model but only the object matched by the source pattern element can be updated. That is, ATL does not support the update those objects matched in the `using` block. However, the matched object (and its contents) can be deleted using the statement `drop` (in the output pattern of a rule). Moreover, the naming conventions `__ref` and `__unset` and their semantics for applying updates to source object variables are ignored by ATL. These naming conventions are used by FMA-ATL to unset references, which cannot be done in ATL.

Regarding system verification, ATL in-place semantics is not sufficient for system specification, as explained in the introduction. To illustrate the unsoundness and the incompleteness of an ATL specification using ATL2010 in-place semantics (w.r.t. the intended behaviour of the system), we consider the scenario of the running example where the same element `'b'` is inserted twice in the singleton list of Fig. 1. We have modified the rule `Append` as a workaround for the problems stated above, that is to help ATL apply updates to the matched object only. The main change in the state model of Fig. 1 involves the declaration of a reference `previous` as opposite to `next`. The new rule `Append`, shown in Fig. 5, captures the intended behaviour of the original rule: a new cell is appended to the list if it has not been found (the appender is marking the last element of the list, which has a different value). To apply the transformation, ATL computes the matches, enabling the rule for both objects `Append`, with ids `1` and `2`. When the transformation is executed, the first application of the rule `Append` inserts the new cell with id `4`. This should disable the match of the rule for object `1`.

```
rule Append {
  from a1 : append!Append (
    a1.active and
    a1.this.next.oclIsUndefined()
    and a1.this.value <> a1.x
) to a1__ref : append!Append (
    active <- false,
    return <- true,
    x <- ''
  ),
  c2 : append!Cell (
    value <- a1.x,
    previous <- a1.this,
    list <- a1.this.list
  )
}
```

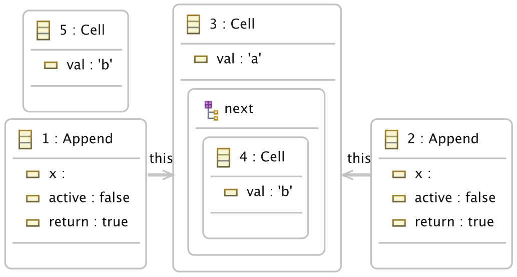


Fig. 5. Modified rule `Append` (left) and resulting state (right) from state of Fig. 1.

However, as the engine is blindly applying the pre-computed matches, a new cell with id `5` is inserted, leaving cell `4` dangling because of the upper bound of the reference `next`, as shown in the resulting state in Fig. 5. Hence, violating the state property `Shared`.

To consider a witness of incompleteness of an ATL2010 specification using in-place semantics w.r.t. the system behaviour, we look at the rule `Return`, which is enabled for object `1` after the first application of `Append`. However, the execution path of Fig. 4, where rule `Return` is applied before the application of `Append` to object `1`, is obliterated for the same reason and this behaviour is not captured by the ATL2010 in-place semantics of the system specification.

5.2 Comparison

Table 1 shows a comparison of features of ATL2010 in-place semantics with two ATL-based specification languages, namely SimpleGT [12] and FMA-ATL. To give a broader view of the contributions, we have also included Henshin [1], Groove [7] and e-Motions [5,11], which are also based on EMF (either directly or indirectly) and provide rule-based languages both for modelling and for verifying EMF-based systems.

We classify our comparison under two main dimensions: specification and verification. Question marks are inserted wherever definite information could not be found to sustain the claim. From a *system specification* point of view, we consider the type *concrete syntax* used for specifying systems, that is using the ATL language, abstract syntax (object diagrams or similar), domain-specific modelling language (DSML) or other; the language for specifying queries; their support for *negative-application conditions* (NACs); whether *updates* can be applied to *several objects* matched in the query part of the rule; *control* mechanisms to handle the application of rules, for example application of rules as long as possible (*alap*), only one match per rule *unique*, *arbitrary* selection of the rule to be applied, *rule priorities*, a dedicated *control language*, or other scheduling policies; whether rule application *amalgamation* is supported by using mechanisms that group several transitions in one single transition; strategies available to explore

Table 1. Comparison of system specification languages for EMF-based systems.

Features	FMA-ATL in-place	ATL2010 in-place	SimpleGT	Groove	Henshin	e-Motions
System specification						
Concrete syntax	textual (ATL)	textual (ATL)	textual (other)	graphical (abstract)	graphical (abstract)	graphical (DSML)
Query language	mOdCL	ATL-OCL	SimpleOCL	graph patterns	graph patterns	graph patterns, mOdCL
NACs	OCL (filter)	OCL (filter)	✓	✓	✓	✓
Updates	✓	✗	✓	✓	✓	✓
Control	alap	unique	alap? unique	arbitrary priorities control lang	alap priorities	round-robin
Amalgamation	✓	✗	✗	✓	✓	✗
Rule inheritance	✗	✓	✓	✗	✗	✗
Non-determinism	✓	✗	✓	✓	✓	✓
State-space generation	BFS	✗	✗	DFS, BFS linear	BFS?	BFS
Property specification and verification						
Language	ATL helpers	✗	✗	graphs	OCL	Maude
Model checking	bound. inv, LTL	✗	✗	bound. inv, LTL, CTL	bound.? inv, qualitative probabilistic	bound. inv, LTL statistical

the *state space*, usually depth-first search (DFS), breadth-first search (BFS) or linear; and whether the specification language can model *non-determinism*.

Regarding *verification*, we focus on the *language* used to specify state properties and on model checking techniques supported. Additionally: Groove provides mechanisms for symmetry reduction; Henshin[4] provides support for qualitative model checking with CADP and mCRL2, and stochastic and probabilistic model checking with PRISM; and e-Motions system specifications can model both real-time systems and stochastic systems, the latter class of models can be analysed with statistical model checking using PVeStA.

FMA-ATL, Groove and Henshin support amalgamation mechanisms to reduce the state space. In particular, FMA-ATL achieves this by using lazy rules [2]. However, these tools, together with e-Motions, do no provide support for rule inheritance. By using ATL as front-end language in FMA-ATL, the ecosystem of tools available both for developing ATL transformations (e.g. IDE support, parser) and for analysing them can be reused for facilitating the correct definition of ATL transformations/specifications. Conversely, our tool contributes to that ecosystem as well.

[4] http://wiki.eclipse.org/Henshin/State_Space_Tools.

6 Conclusions

Verification of model-based software systems have normally been studied with in-place graph transformation and, up to now, ATL has not been used for this purpose. In this work, we have discussed several drawbacks that hamper the use of the current ATL in-place semantics for modelling and verifying EMF-based systems. In particular, we illustrated why such system specifications are potentially incomplete and unsound w.r.t. the intended behaviour of a system for verification purposes by using a representative example form the literature.

We presented a new in-place semantics for ATL by augmenting FMA-ATL's semantics with a new scheduler rule, by enabling ATL as its front-end language and by linking Maude's verification techniques to ATL. FMA-ATL thus enables the use of ATL for specifying, simulating and verifying both deterministic and non-deterministic systems.

Acknowledgements. The author would like to thank Frédéric Jouault and Massimo Tisi for insightful discussions on the semantics of ATL, and the anonymous reviewers for their observations, which helped improve this work greatly.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_9
2. Boronat, A.: Experimentation with a big-step semantics for ATL model transformations. In: Guerra, E., van den Brand, M. (eds.) ICMT 2017. LNCS, vol. 10374, pp. 3–18. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61473-1_1
3. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
4. Cuadrado, J.S., Guerra, E., de Lara, J.: Uncovering errors in ATL model transformations using static analysis and constraint solving. In: ISSRE, pp. 34–44. IEEE Computer Society (2014)
5. Durán, F., Moreno-Delgado, A., Álvarez-Palomo, J.M.: Statistical model checking of e-motions domain-specific modeling languages. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 305–322. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_18
6. Eker, S., Meseguer, J., Sridharanarayanan, A.: The maude LTL model checker. Electr. Notes Theor. Comput. Sci. **71**, 162–187 (2002)
7. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. STTT **14**(1), 15–40 (2012)
8. Roldán, M., Durán, F.: The mOdCL evaluator: Maude + OCL (2013). http://maude.lcc.uma.es/mOdCL/. Accessed 3 Mar 2016
9. Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: a comparison of two approaches. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 226–241. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_17

10. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Refining Models with Rule-based Model Transformations. Research Report RR-7582, March 2011
11. Troya, J., Rivera, J.E., Vallecillo, A.: Simulating domain specific visual models by observation. In: SpringSim, p. 128. SCS/ACM (2010)
12. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 623–637. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_46

Tool Demonstration Papers

Scalable Queries and Model Transformations with the Mogwaï Tool

Gwendal Daniel[1(✉)], Gerson Sunyé[2], and Jordi Cabot[1,3]

[1] Internet Interdisciplinary Institute (IN3), Universitat Oberta de Catalunya (UOC), Barcelona, Spain
gdaniel@uoc.edu

[2] LS2N, Université de Nantes, Nantes, France
gerson.sunye@ls2n.fr

[3] ICREA, Barcelona, Spain
jordi.cabot@icrea.cat

Abstract. Scalability of modeling frameworks has become a major issue hampering MDE adoption in the industry. Specifically, scalable model persistence, as well as efficient query and transformation engines, are two of the key challenges that need to be addressed to enable the support for very large models in current applications. In this paper we demonstrate Mogwaï, a tool designed to efficiently compute queries and transformations (expressed in OCL and ATL) over models stored in NoSQL databases. Mogwaï relies on a translational approach that maps constructs of the supported input languages to Gremlin, a generic NoSQL query language, and a model to datastore mapping allowing to compute the generated query on top of several datastores. The produced queries are computed on the database side, benefiting of all its optimizations, improving the execution time and reducing the memory footprint compared to standard solutions. The Mogwaï tool is released as a set of open source Eclipse plugins and is fully available online.

Keywords: MDE · Scalability · OCL · ATL · Model query
Model transformation

1 Introduction

Existing empirical assessments from industrial companies adopting MDE [14] point to the limited support for managing large models as one of the factors limiting the success of MDE in industrial MDE processes. Indeed, existing modeling solutions were primary designed to handle simple, human-based modeling activities, and existing technical solutions are not designed to handle large models (potentially generated using model driven reverse engineering techniques [2]) commonly used nowadays. In particular, several studies have reported the scalability issues of the Eclipse Modeling framework (the *de-facto* standard framework for building modeling tools in the Eclipse community) and its default serialization mechanism XMI.

These limitations have led to the creation of several scalable model persistence frameworks built on top of different types of databases [1,4,7] combined with advanced mechanisms such as application-level caches [13] and *lazy-loading*. While this new generation of model persistence techniques has globally improved the support for managing large models, they are partial solutions to the scalability problem in current modeling frameworks. In its core, all frameworks are based on the use of low-level model handling APIs that are focused on manipulating individual model elements and do not provide support for generic model query and transformation computation. This approach is clearly inefficient because (i) the API granularity is too fine-grained to benefit from the advanced query capabilities of the backend and (ii) an important time and memory overhead is necessary to construct navigable intermediate objects that are needed by the modeling API.

To overcome this situation, we developed Mogwaï, a scalable query and transformation framework for large models. Mogwaï consists of a translation component that maps model queries and transformations (expressed in OCL [11] and ATL [8]) into expressions of a graph traversal language, Gremlin [12], a multi-database graph traversal query language we use as our output language. Generated queries are then directly computed on the database side, bypassing the standard modeling API. This avoids the above mentioned problems and significantly improves the overall performance.

This paper complements our existing work [3,5] by introducing its modular architecture, additional tool implementation details, and includes a unified *ModelDatastore* component that allows to access multiple datastores transparently for both query and transformation computations.

The rest of the paper is organized as follows: Sect. 2 gives an overview of the Mogwaï infrastructure, Sect. 3 presents the architecture of our tool and its query processing engine, and Sect. 4 presents the tool implementation. Finally, Sect. 5 summarizes the key points of the paper.

2 Framework Overview

Figure 1 shows an overview of the Mogwaï framework that creates Gremlin scripts from input model queries and transformations. An initial *Model Query* or *Transformation* is parsed and sent to a *Translation Engine*, that selects the translation to apply and performs a systematic mapping of the input expressions' to Gremlin constructs. These constructs are then assembled into a *Gremlin Script* that is sent to the database for computation.

The Mogwaï *Translation Engine* relies on a *Model Datastore Definition* to produce the output gremlin script. This generic library provides an abstraction layer that decouples the computation from the low-level database access by adding modeling primitives to manipulate natively the data representing the model. As a result, the generated *Gremlin Script* is not tailored to a specific data store, and can be parametrized with a *Model Datastore Implementation*, that wraps the concrete *Model Datastore* to use (i.e. the backend storing the model).

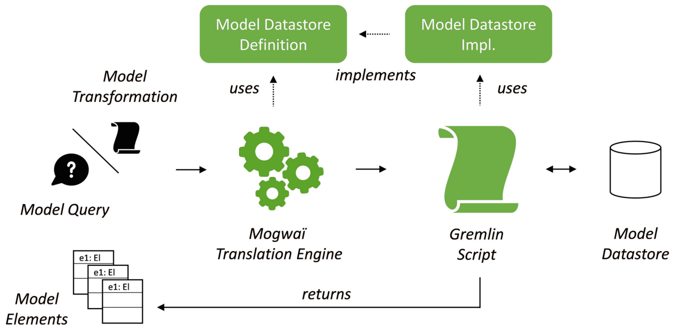


Fig. 1. Mogwaï infrastructure

This architecture, originally defined for the Gremlin-ATL engine [3], has been integrated in the OCL engine [5] to allow to query models stored in multiple types of data storage solutions, and can be easily extended to support additional backends.

Internally, the framework defines two model-to-model transformations: *OCL2Gremlin*, that handles model queries expressed using the OCL language [11], and *ATL2Gremlin*, that translates model transformations expressed in ATL [8]. Note that the modular architecture of the framework allows to define additional translations to support alternative query and transformation solutions such as EOL [9] or QVT [10].

The generated scripts can be returned to the modeler and used as stored procedures to execute in the future, or directly computed with a specific implementation of the *Model Datastore* library. Finally, the returned elements from the computation (if any) are reified into regular model elements thanks to the *Model Datastore* implementation.

Compared to existing query frameworks, Mogwaï does not rely on the default modeling API to compute model queries and transformations. In general, API based frameworks translate queries and transformations into a sequence of low-level API calls, which are then performed one after another on the persistence layer. While this approach has the benefit to be compatible with every API-based applications, it does not take full advantage of the database structure and query optimizations. Furthermore, each object fetched from the database has to be reified to be navigable, even if it is not going to be part of the end result. Therefore, the execution time and memory consumption of the API-based solutions strongly depends on the number of intermediate objects fetched from the database.

3 Architecture

Figure 2 describes the internal structure of the Mogwaï framework. The *QueryProcessor* is the core of the engine: it provides the *process* method, that

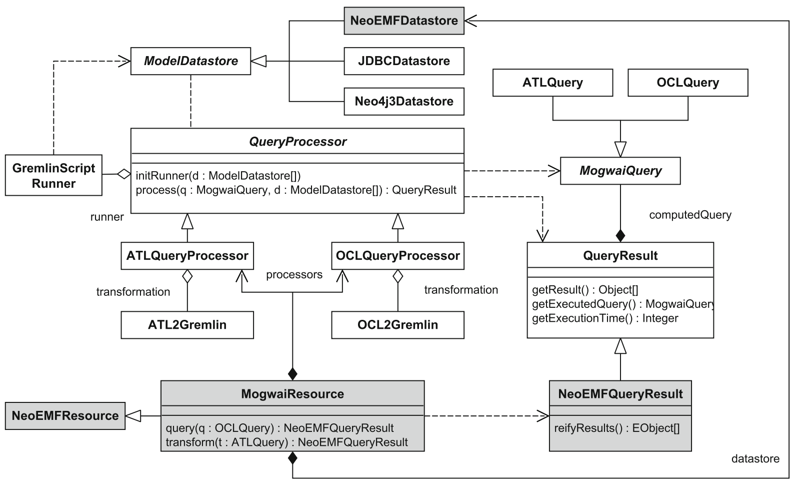


Fig. 2. Mogwaï internal structure

takes as its input a *MogwaiQuery* and a set of *ModelDatastores*, and returns a *QueryResult* containing the result of the computation and additional monitoring information (such as the computed *MogwaiQuery* and the raw query execution time).

The *QueryProcessor* relies on an internal *GremlinScriptRunner* that provides utility methods to setup a Gremlin environment and execute queries. Note that the tool provides two implementations of the abstract *QueryProcessor*: the first one, *ATLProcessor*, is dedicated to ATL transformation computation, and the second one, *OCLProcessor*, handles OCL queries. Both processors rely on an internal model to model transformation responsible for mapping the constructs of the input languages to Gremlin. Note that this modular architecture could be easily extended to support alternative query and transformation languages.

The Mogwaï's architecture is not tailored to a specific backend or model persistence technology, and can be used on top of any *ModelDatastore* implementation. However, the tool also provides an advanced integration into the NeoEMF platform (light-grey boxes) that speeds-up query computation and improves the tool's integration with existing EMF-based application by returning EMF-compatible objects. The *MogwaiResource* class extends the NeoEMF one with a simple API defining the `query` and `transform` methods. This resource embeds a set of *QueryProcessors* as well as a preset *ModelDatastore* implementation targeting the native API of the database storing the model. Query and transformations executed through the *MogwaiResource* return *NeoEMF-QueryResults*, that contain database records that can be reified into navigable EMF elements if needed. Note that resulting model elements are created only from the results of the Gremlin script execution, removing the memory overhead implied by intermediate objects created during EMF-based computations.

4 Implementation

The Mogwaï tool is implemented as a set of open-source Eclipse plugins released under the EPL license. Source code and benchmark materials are fully available in the project's GitHub repository[1]. An Eclipse update site containing the last stable version of the framework is also available online[2].

The OCL engine relies on Eclipse MDT OCL [6] to parse the input queries, and the produced OCL models constitute the input of a set of 70 ATL [8] transformation rules and helpers implementing the mapping and the transformation process presented in detail in our previous work [5].

The ATL engine presents a similar architecture, and relies on the ATL parser to create a model from the transformation to compute. This transformation model is then sent to a high-order transformation mapping ATL constructs to Gremlin [3] (represented as a set of 80 rules and helpers).

5 Conclusion

We have showcased Mogwaï, a tool that generates Gremlin scripts from model queries and transformations in order to maximize the benefits of using a NoSQL backend to store and manipulate large models. Gremlin scripts are created using a set of model-to-model transformations, and are parametrized with a specific *Model Datastore*, enabling their computation over a variety of backends compatible with the Gremlin language. The Mogwaï approach allows to bypass the existing modeling framework's API, improving the performance of query and transformation computations both in terms of execution time and memory consumption [3,5]. The tool development roadmap for Mogwaï includes adding support for more types of NoSQL backends, like document-oriented and column databases by providing the necessary translations from ATL and OCL to their native languages.

Appendix A Demonstration Overview

The demonstration presents two typical use cases where the Mogwaï framework significantly improves the execution time and memory consumption of an application computing OCL queries and ATL transformations on top of large models.

First, we briefly introduce the `sample` model, that is a real-world model obtained by applying model driven reverse engineering techniques on an existing code base. The manipulated model contains around 80 000 elements representing a Java application. Figure 3 shows an excerpt of the metamodel describing the `sample`. Note that the complete metamodel can be found in the MoDisco repository[3].

[1] https://github.com/atlanmod/Mogwai.
[2] atlanmod.github.io/Mogwai.
[3] http://git.eclipse.org/c/modisco.

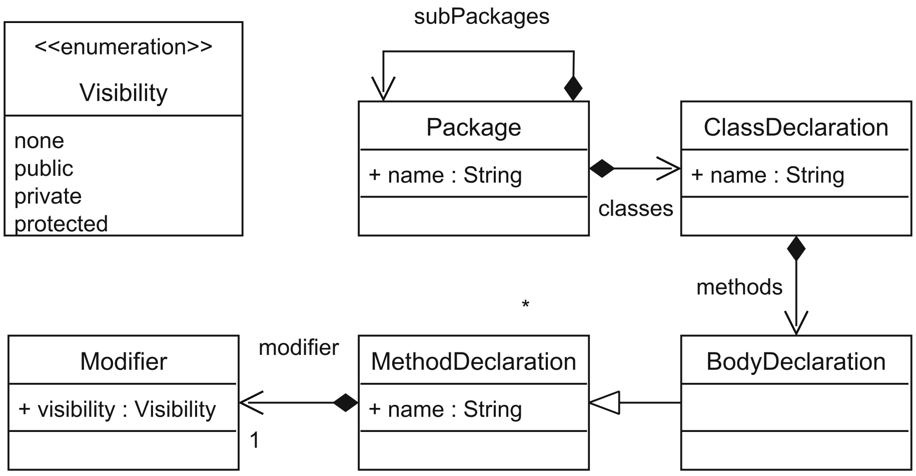


Fig. 3. Excerpt of the MoDisco Java metamodel

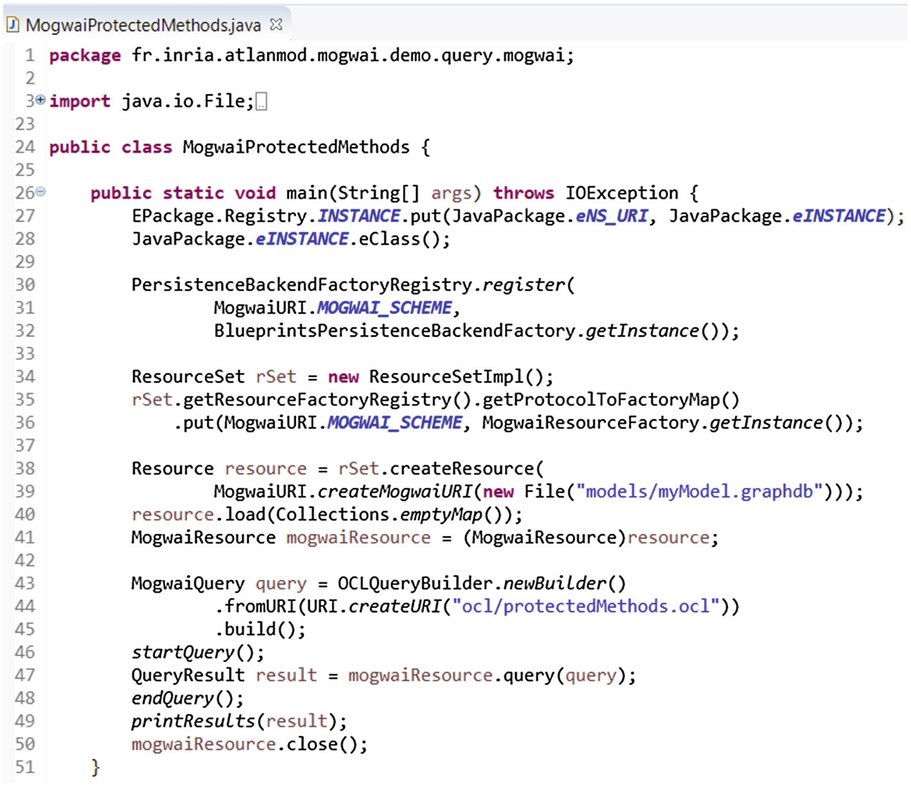
Then, we present a set of OCL queries to compute against this model. An example of such query is provided in Listing 1.1, that describes the `protectedMethod` query, which finds in the model all the *MethodDeclarations* that have a protected *Modifier*. We show in the demonstration how to initialize and configure the Mogwaï engine to translate and run the query from an existing Java application (Fig. 4). In parallel, we show how the query is computed using the regular Eclipse MDT-OCL interpreter embedded with Eclipse, and emphasize the differences.

Listing 1.1. Sample OCL Query

```
context ClassDeclaration
def: protectedMethods : Sequence(BodyDeclaration) =
ClassDeclaration.allInstances()->
 collect(bodyDeclarations)->
  select(each | each.oclIsTypeOf(MethodDeclaration))->
   select(each | not(each.modifier.oclIsUndefined()))->
 select(each | each.modifier.visibility = VisibilityKind::protected)->
 asSequence()
```

Then, we introduce a simple model-to-model transformation defined with the ATL language to compute on top of the `sample` model (Listing 1.2). This transformation extracts all the *ClassDeclaration* instances from the input model and maps them to the *Table* construct of the output metamodel, and sets a unique *key* that allows to identify a *ClassDeclaration* instance. A second rule is responsible of transforming each *MethodDeclaration* into a *Column* representing the number of calls to the method.

In the demonstration, we show the required steps to initialize the Mogwaï engine with the transformation and compute it against the database storing the model (Fig. 5). In addition, we show how the output model can be stored in another data representation using a different *Model Mapping Implementation*. We also compare the execution time of computing the transformation using the



MogwaiProtectedMethods.java

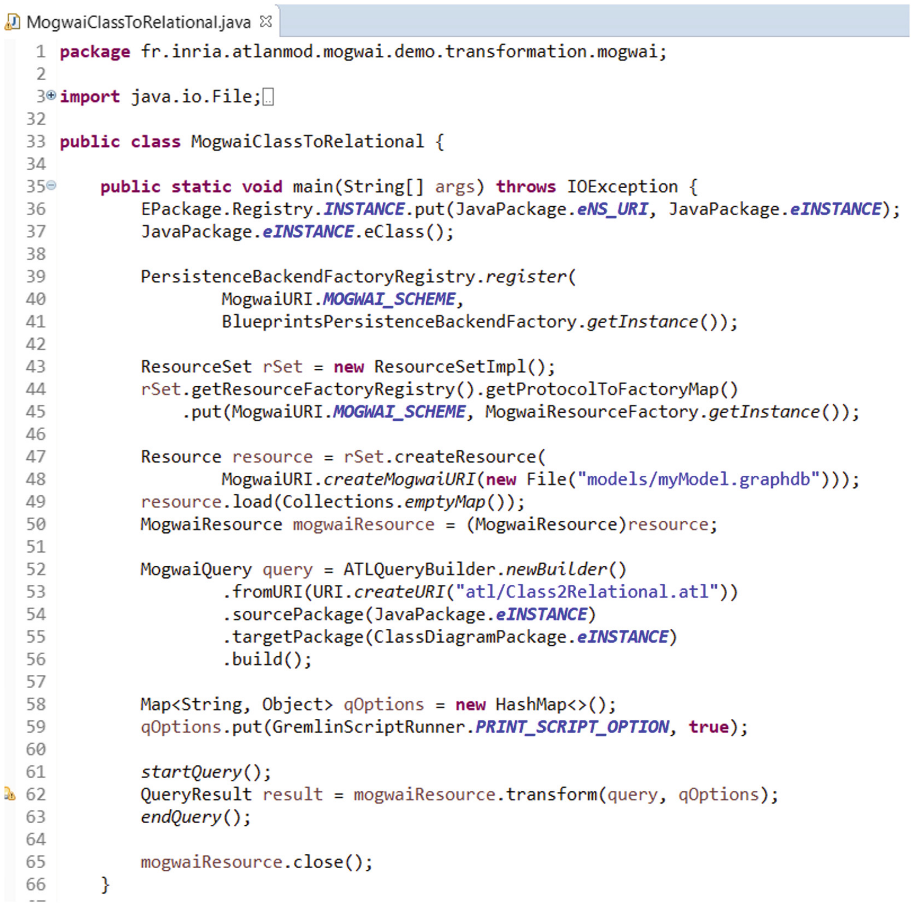
```
package fr.inria.atlanmod.mogwai.demo.query.mogwai;

import java.io.File;

public class MogwaiProtectedMethods {

    public static void main(String[] args) throws IOException {
        EPackage.Registry.INSTANCE.put(JavaPackage.eNS_URI, JavaPackage.eINSTANCE);
        JavaPackage.eINSTANCE.eClass();

        PersistenceBackendFactoryRegistry.register(
                MogwaiURI.MOGWAI_SCHEME,
                BlueprintsPersistenceBackendFactory.getInstance());

        ResourceSet rSet = new ResourceSetImpl();
        rSet.getResourceFactoryRegistry().getProtocolToFactoryMap()
            .put(MogwaiURI.MOGWAI_SCHEME, MogwaiResourceFactory.getInstance());

        Resource resource = rSet.createResource(
                MogwaiURI.createMogwaiURI(new File("models/myModel.graphdb")));
        resource.load(Collections.emptyMap());
        MogwaiResource mogwaiResource = (MogwaiResource)resource;

        MogwaiQuery query = OCLQueryBuilder.newBuilder()
                .fromURI(URI.createURI("ocl/protectedMethods.ocl"))
                .build();
        startQuery();
        QueryResult result = mogwaiResource.query(query);
        endQuery();
        printResults(result);
        mogwaiResource.close();
    }
```

Fig. 4. Running OCL queries with Mogwaï

regular ATL engine with Mogwaï, and show that using our approach can bring significant improvements in terms of execution time.

Listing 1.2. Sample ATL Transformation

```
module Class2Relational;
create OUT : RelationalMM from IN : Java;

rule Class2Table {
  from
    c : Java!ClassDeclaration
  to
    out : RelationalMM!Table (
      name ← c.name,
      col ← Sequence{key}->union(c.bodyDeclarations
        ->select(b | b.oclIsTypeOf(Java!MethodDeclaration))),
      key ← key
    ),
    key : RelationalMM!Column (
      name ← 'objectId',
      type ← keyType
    ),
    keyType : RelationalMM!Type (
        name ← 'Integer'
      )
  }

  rule Method2Column {
    from
      m : Java!MethodDeclaration
    to
      out : RelationalMM!Column (
        name ← m.name + 'CallCount',
        type ← type
      ),
      type : RelationalMM!Type (
        name ← 'Integer'
      )
  }
```



MogwaiClassToRelational.java

```java
package fr.inria.atlanmod.mogwai.demo.transformation.mogwai;

import java.io.File;

public class MogwaiClassToRelational {

    public static void main(String[] args) throws IOException {
        EPackage.Registry.INSTANCE.put(JavaPackage.eNS_URI, JavaPackage.eINSTANCE);
        JavaPackage.eINSTANCE.eClass();

        PersistenceBackendFactoryRegistry.register(
                MogwaiURI.MOGWAI_SCHEME,
                BlueprintsPersistenceBackendFactory.getInstance());

        ResourceSet rSet = new ResourceSetImpl();
        rSet.getResourceFactoryRegistry().getProtocolToFactoryMap()
            .put(MogwaiURI.MOGWAI_SCHEME, MogwaiResourceFactory.getInstance());

        Resource resource = rSet.createResource(
                MogwaiURI.createMogwaiURI(new File("models/myModel.graphdb")));
        resource.load(Collections.emptyMap());
        MogwaiResource mogwaiResource = (MogwaiResource)resource;

        MogwaiQuery query = ATLQueryBuilder.newBuilder()
                .fromURI(URI.createURI("atl/Class2Relational.atl"))
                .sourcePackage(JavaPackage.eINSTANCE)
                .targetPackage(ClassDiagramPackage.eINSTANCE)
                .build();

        Map<String, Object> qOptions = new HashMap<>();
        qOptions.put(GremlinScriptRunner.PRINT_SCRIPT_OPTION, true);

        startQuery();
        QueryResult result = mogwaiResource.transform(query, qOptions);
        endQuery();

        mogwaiResource.close();
    }
```

Fig. 5. Running ATL queries with Mogwaï

Finally, the key points of the tool will be summarized and some remarks on the integration into existing modeling application will be provided. All the presented examples and models will be publicly available on the Mogwai GitHub repository. In addition, a video summarizing the key points of the demonstration is available online at https://youtu.be/_nTBPJMVRQY.

References

1. Barmpis, K., Kolovos, D.: Hawk: towards a scalable model indexing architecture. In: Proceedings of the 1st BigMDE Workshop, pp. 6–9. ACM (2013)
2. Bruneliere, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: a model driven reverse engineering framework. Inf. Softw. Technol. **56**(8), 1012–1032 (2014)
3. Daniel, G., Jouault, F., Sunyé, G., Cabot, J.: Gremlin-ATL: a scalable model transformation framework. In: Proceedings of the 32nd ASE Conference, pp. 462–472. IEEE (2017)
4. Daniel, G., Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., Cabot, J.: NeoEMF: a multi-database model persistence framework for very large models. Sci. Comput. Program. **149**, 9–14 (2017)
5. G. Daniel, G. Sunyé, and J. Cabot. Mogwaï: a framework to handle complex queries on large models. In: Proceedings of the 10th RCIS Conference, pp. 225–237. IEEE (2016)
6. Eclipse Foundation. MDT OCL (2018). http://www.eclipse.org/modeling/mdt/?project=ocl
7. Eclipse Foundation. The CDO Model Repository (CDO) (2018). http://www.eclipse.org/cdo/
8. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Programm. **72**(1), 31–39 (2008)
9. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon object language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006). https://doi.org/10.1007/11787044_11
10. OMG. QVT Specification (2017). http://www.omg.org/spec/QVT
11. OMG. OCL Specification (2018). http://www.omg.org/spec/OCL
12. Tinkerpop. The Gremlin Language (2018). www.gremlin.tinkerpop.com
13. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-INCQUERY: an integrated development environment for live model queries. Sci. Comput. Program. **98**, 80–99 (2015)
14. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Softw. **31**(3), 79–85 (2014)

NMF: A Multi-platform Modeling Framework

Georg Hinkel(✉)

FZI Research Center of Information Technologies (FZI),
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
hinkel@fzi.de

Abstract. For its promises in terms of increased productivity, Model-driven engineering (MDE) is getting applied increasingly often in both industry and academia. However, most tools currently available are based on the Eclipse Modeling Framework (EMF) and hence based on the Java platform whereas tool support for other platforms is limited. This leads to a language and tool adoption problem for developers of other platforms such as .NET. As a result, few projects on the .NET platform adopt MDE. In this paper, we present the .NET Modeling Framework (NMF), a tool set for model repositories, model-based incrementalization, model transformation, model synchronization and code generation that is now available for a multitude of different operating systems, including Windows, Linux, Android, iOS and Mac. The framework makes intensive use of the C# language as host language for model transformation and synchronization languages, whereas the model repository serialization is compatible with EMF. This solves the language adoption problem for C# programmers and creates a bridge to the EMF platform.

1 Introduction

Model-driven engineering (MDE) is getting applied increasingly often both in industry and academia. Dedicated support to use models for analysis or transformation purposes reduces manual development efforts as repetitive infrastructure code can be reused. Most of the existing tools that support MDE are currently based on the Java platform. As a consequence, legacy software built on other platforms can hardly be reused.

Furthermore, MDE is increasingly applied on mobile platforms [1] where traditional tools such as Eclipse are difficult to operate and alternatives are necessary. Ideally, such alternative modeling environments should support as many platforms as possible to reduce the code duplication in the support for multiple platforms.

In this paper, we present the .NET Modeling Framework (NMF), a framework of libraries, tools and languages to support model-driven engineering on the .NET platform. The framework is dedicated to process existing models through analysis, transformation and synchronization. NMF contains tools to

A. Rensink and J. Sánchez Cuadrado (Eds.): ICMT 2018, LNCS 10888, pp. 184–194, 2018.
https://doi.org/10.1007/978-3-319-93317-7_10

generate model representations compliant with EMF, supports a model management repository system and allows developers to specify model analyses, model transformations and model synchronizations. To minimize both the language adoption problem and the tool support problem, NMF is entirely based on internal languages that use C# as a host language.

Since December 2017, the runtime libraries of NMF all support the .NET Standard 2.0 and are therefore usable not only in Windows but also on various other platforms such as Linux and Mac through .NET Core[1], but also Android and iOS through the Xamarin platform[2]. In particular, NMF allows to create model-based libraries that can be shared across all of these platforms.

An introductory tutorial for NMF can be found on YouTube[3].

The remainder of this paper is structured as follows: Sect. 2 presents the meta-metamodel used in NMF and discusses serialization. Section 3 explains the support for model repositories and how they are used. Section 4 describes the support for implicit incremental model analyses that is built into NMF. Section 5 introduces the model transformation language NTL. Section 6 shows how the concepts are combined in a language for the synchronization of heterogeneous metamodels. Finally, Sect. 7 concludes the paper.

2 Meta-metamodel

NMF contains its own meta-metamodel called NMeta. NMeta is similar to Ecore but contains dedicated support for type system features widely used on the .NET platform such as structures or events. Furthermore, it also supports an extension mechanism closely related to stereotypes as well as refinements. The semantics of NMeta is clearly defined through a mapping to category theory. Though there is a high semantic overlap with the Essential Meta Object Facility (EMOF) standard, there are also some features that do not have a counterpart in NMeta, in particular factories and generic types.

However, since Ecore is the meta-metamodel most often used in MDE, NMF contains a model transformation from Ecore to NMeta. This transformation is based on the extensible Model Transformation Language NTL (cf. Sect. 5 or [2]) and thus support for other types can be easily added.

The resulting NMeta metamodel is compliant with the original Ecore metamodel if the latter only contains basic structures (packages, classes, attributes and references). Here, compliant means that serialized instances of the original Ecore metamodel can be deserialized with the NMeta metamodel (if no custom XMI handlers are used) and vice versa. In particular, the XMI serialization of the metamodels is equivalent and the NMF serializer uses the same addressing scheme for cross references as the EMF serializer uses for Ecore.

[1] A list of supported linux distributions is available under https://github.com/dotnet/core/blob/master/release-notes/2.0/2.0-supported-os.md.

[2] http://www.xamarin.com/platform.

[3] https://youtu.be/NIMYuwTltVs.

Similar to Ecore, NMeta is bootstrapped and the classes `ModelElement` and `Model` are the only ones with a custom implementation, the implementation of all other classes originate directly from the code generator.

3 Model Repositories

In NMeta, all model elements have both an absolute and a relative URI that allow developers to easily reference model elements in a defined way. The addressing scheme is based on the containment hierarchy where the elements are identified by their identifier or by the collection index. The syntax is the same as used in the EMF serializer to push interoperability to EMF.

NMF is able to resolve URIs from different sources, including files, embedded resources and network streams. To resolve a model, NMF uses a singleton meta repository (which itself is a model repository) where all metamodels are loaded and linked to the implementation, if available. The registration of model representation code is done simply through an assembly annotation that links a namespace to an assembly embedded resource where the metamodel is formally described. Here, assemblies are the components of the .NET component model. When the meta-repository is loaded for the first time, it iterates through the loaded assemblies and finds all metamodels registered, so that a repository is able to load a model just in case the assembly containing the model representation classes is referenced.

4 Model-Based Incrementalization

Again similar to EMF, NMF provides elementary change notifications, offered through the industry standard interfaces `INotifyPropertyChanged` and `INotifyCollectionChanged`. These interfaces are required by many modern user interface libraries, hence the model representation code can directly be used for these techniques.

However, NMF is also able to combine these elementary change notifications to determine when the result of analyses based on a model has changed. Furthermore, an incremental algorithm is inferred to recalculate the analysis upon a model change more efficiently by the implicit introduction and management of buffers to save intermediate results. This incrementalization works online, i.e. the model needs to be kept in memory and changes must be made on the model elements in memory.

The incrementalization has a sound theoretical foundation based on category theory and is implemented in NMF Expressions. NMF Expressions operates on lambda expressions, supported by many .NET languages such as C# and VB.NET in their regular syntax. To realize the incrementalization, the abstract syntax tree is converted into a dynamic dependency graph on a high abstraction level. Changes of the model under analysis are then propagated through the dependency graph, ultimately updating the analysis result.

```
var faultyPositions = from route in routes
  where route.Entry != null && route.Entry.Signal == Signal.GO
  from swP in route.Follows
  where swP.Switch.CurrentPosition != swP.Position
  select swP;
```

Listing 1. Query to find inaccurate switch positions in a collection of routes

As an example, consider the code in Listing 1, taken from the NMF solution of the TTC Train Benchmark [3]. NMF allows the user to specify queries like this in regular C# code with all of the tool support provided for this language and is able to implicitly deduct an incremental evaluation.

The high abstraction level in the dynamic dependency graph is achieved by a manual incrementalization of analysis operators yielding valid results as a consequence of the underlying formalization as a categorial functor. NMF Expressions includes a library of such manually incrementalized operators, including most of the Standard Query Operators (SQO)[4]. As a consequence, developers can specify query analyses conveniently through the query syntax such as used in Listing 1.

5 Model Transformation

To support model transformation, NMF contains the NMF Transformations Language (NTL) [4], an internal model transformation language integrated in C#, reusing the tool support for C# [5]. This transformation language allows to specify extensible rule-based model transformations with explicit dependencies between the transformation rules. The underlying transformation engine is not restricted to NMeta models as input or output models such that also arbitrary CLR objects can be transformed where the CLR denotes the .NET virtual machine, similar to the JVM in Java.

Model transformations in NTL are essentially classes whose transformation rules are inferred by the public nested classes. These are encoded also as separate classes that inherit from a set of generic base classes and the generic type parameters specify the source and target model elements. These transformation rule classes may override a method to define their dependencies. Inside this method, transformation rules may define dependencies to other transformation rules, their instantiation or patterns that declaratively specify when the transformation rule should be called. Other than that, the transformation rules may override a method that is called to initialize the transformation rule result. Similar to ATL, NTL also allows transformations to be based on other transformation rules overriding some of their transformation rules. This technique is called superimposition in ATL [6], in NTL it is called transformation rule inheritance as it is realized in inheriting the transformation rule classes.

6 Model Synchronization

Based on NTL and NMF Expressions, NMF also contains a language to synchronize models of heterogeneous metamodels, named NMF Synchronizations

[4] http://msdn.microsoft.com/en-us/library/bb394939.aspx.

[7]. Like NTL, it is also implemented as an internal DSL so that developers can familiarize quickly. This synchronization language is able to support 18 different operation modes out of a single specification: One may choose between three different change propagation modes (none, one-way and two-way) and six different directions (left-to-right and right-to-left in three different variants each).

Similar to NTL, a synchronization rule in NMF Synchronizations is represented by a class, inferring the synchronization rules by the public nested classes. The synchronization rules each define an isomorphism between the classes they are to synchronize, referred to as left-hand-side (LHS) and right-hand-side (RHS) class. These classes are passed as generic type parameters.

7 Conclusion

In this paper, we have given an overview on NMF, a framework to support model-driven engineering on the .NET platforms. Through the support of .NET Standard, NMF is available on most modern platforms, including Windows, Linux, Android, iOS and Mac. The framework is largely compatible with EMF such that EMF models (metamodels and instance models) can be reused. Further, the framework provides tools to generate model representation code and analyze, transform and synchronize the models, also incrementally.

A Tutorial of NMF

The following instructions will describe how to set up a project with NMF and create a model transformation from state machines to Petri nets. Although the tutorial is specifically written for a usage in Visual Studio, the tutorial can be adapted to any IDE on the .NET platform. The tutorial also assumes that you have already started to create a metamodel and some instances of it in EMF, i.e. Eclipse.

If you get stuck at any point, there is a ready-made solution available on GitHub[5] that can be just downloaded and tried. Furthermore, there is a YouTube video available[6] demonstrating creating a new project, loading, altering and saving a model.

A.1 Create a Project

NMF is a framework that can be easily installed through the NuGet Package-manager[7]. Therefore, first create a new project. In Visual Studio, click on *File* →*New*→ *Project*, select a C# console application as project type and name it as you wish, though in the remainder we will assume the name *NMFDemo*.

[5] https://github.com/NMFCode/NMFDemo.
[6] https://youtu.be/NIMYuwTltVs.
[7] http://www.nuget.org.

Note that NMF is generally not restricted to console applications nor to C#, you can use it in any .NET project.

To import NMF, go to *Tools→NuGet Package-manager→Manage NuGet packages for this project* and search for *NMF*. You should find the package **NMF-Basics**. Install it by hitting the *Install* button while your project is selected.

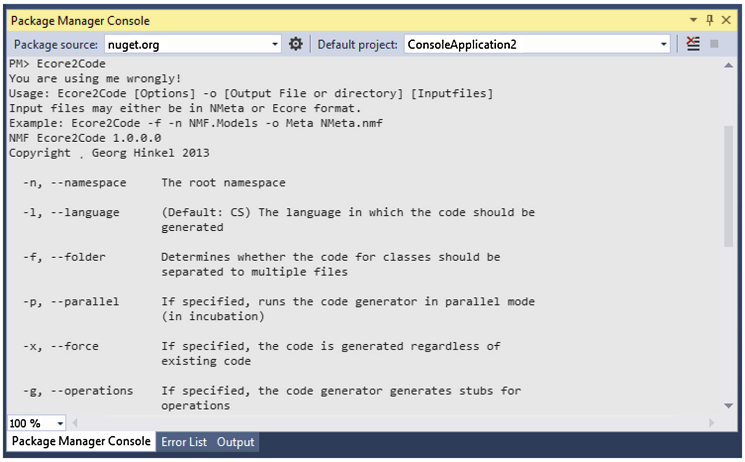
Alternatively, there is also a NuGet console at the bottom, where you can install NMF as follows:

```
PM> Install-Package NMF-Basics
```

NuGet will download the package for you together will all of its dependencies and add all the contained libraries as references into the current project. There is no strict 1:1-mapping from NuGet packages to libraries so there are multiple libraries being installed that may be not needed.

A.2 Import Metamodels from Ecore

Metamodels are at the core of any model-driven development process. Thus, as a first step, we will generate code in order to be able to load any models for a given metamodel in our .NET application. For this, the NuGet package NMF-Basics contains the console application Ecore2Code. After a restart of Visual Studio, NuGet will automatically add Ecore2Code to the Path variable used inside Visual Studio, so you can just use the NuGet Package-manager console. If run without any arguments, this application prints a help information showing its correct usage (cf. Fig. 1).



Package source: nuget.org
Default project: ConsoleApplication2

```
PM> Ecore2Code
You are using me wrongly!
Usage: Ecore2Code [Options] -o [Output File or directory] [Inputfiles]
Input files may either be in NMeta or Ecore format.
Example: Ecore2Code -f -n NMF.Models -o Meta NMeta.nmf
NMF Ecore2Code 1.0.0.0
Copyright , Georg Hinkel 2013

  -n, --namespace     The root namespace

  -l, --language      (Default: CS) The language in which the code should be
                      generated

  -f, --folder        Determines whether the code for classes should be
                      separated to multiple files

  -p, --parallel      If specified, runs the code generator in parallel mode
                      (in incubation)

  -x, --force         If specified, the code is generated regardless of
                      existing code

  -g, --operations    If specified, the code generator generates stubs for
                      operations
```

Fig. 1. The console application Ecore2Code to generate model representation code.

Now, use this tool to generate the code for the state machine metamodel and the Petri net metamodel. You can download these metamodels from our examples project[8]. First copy the metamodels into your project folder, then generate the code for them. The complete commandline for the latter is as follows:

```
PM> Ecore2Code -f -n NMFDemo.Metamodels -m fsm.nmf -o Metamodels\FiniteStateMachines fsm.ecore
PM> Ecore2Code -f -n NMFDemo.Metamodels -m pn.nmf -o Metamodels\PetriNets pn.ecore
```

The generated code now has to be added to your project. Thus, first display all files in the projects folder by clicking on *Show All Files* in the project explorer, then include the generated folder *Metamodels* and the generated NMeta metamodels into your project (right-click and *Include In Project*).

As soon as the generated code is added to the project, it is already possible to programatically create and save models. However, the metamodel is not yet registered and thus no models can be loaded. To register the metamodel, we first need to include the NMeta metamodel in the assembly as an embedded resource and then register the metamodel. To make the metamodel an embedded resource, simply change its *Build Option* to *Embedded Resource* in the properties view while the metamodel is selected.

The metamodel registration is done through an assembly-wide attribute, which can be specified anywhere in the project. The typical place for this registration, however, would be the *AssemblyInfo.cs* file in the properties folder. At the top of this file, add the following two lines:

```
[assembly: NMF.Models.ModelMetadata("http://github.com/NMFCode/Examples/FiniteStateMachines", "NMFDemo.
      fsm.nmf")]
[assembly: NMF.Models.ModelMetadata("http://github.com/NMFCode/Examples/PetriNets", "NMFDemo.pn.nmf")]
```

This is all there is, even if you compile your project not as an executable but as a reusable library. As a reason, when loading the serializer, NMF looks for these attributes in all assemblies referenced by the executing assembly and loads any metamodel registrations it can get.

A.3 Loading a Model

In NMF, models are loaded by resolving their URI in a model repository. If the repository does not contain a model with the given URI, then the model is automatically loaded into the repository, provided NMF is able to locate it. Repositories are closed under cross-reference, meaning that all references to other model elements are always resolved within the repository or its parent repository.

To create a repository, we simply need to create an object of type `ModelRepository`. With the default configuration, this repository is able to deserialize any models conforming to metamodels registered in referenced assemblies, as all repositories implicitly use the meta repository where the metamodels are loaded into.

[8] https://github.com/NMFCode/NMFDemo.

```
var repository = new ModelRepository();
var model = repository.Resolve("Example.fsm");
var fsm = model.RootElements[0] as FiniteStateMachine;
```

Listing 2. Loading Models in NMF

For example, the code needed to load a model from the file *Example.fsm*[9] representing a small order process is depicted in Listing 2. Add these lines to the main method. You can now launch the application and validate that the model can be loaded successfully.

A.4 Incrementalization

The generated model representation classes for the metamodel support change notifications through the .NET de-facto standard interfaces `INotifyPropertyChanged` and `INotifyCollectionChanged`. Thus, the generated classes raise events whenever some properties have been changed or elements have been added to or removed from collections. NMF is able to combine these elementary change notifications to deduct when the value for a combined expression has changed.

For example, let us analyze hubs in the finite state machines, i.e. states that have the maximum incoming transitions. A set of such states can be deducted through the analysis depicted in Listing 3.

```
var stateHubs = from s in fsm.States
    where s.Incoming.Count == fsm.States.Max(s2 => s2.Incoming.Count)
    select s.Name;
```

Listing 3. Analyzing which states are hubs

Verify that the variable `stateHubs` is of type `IEnumerable<string>`, i.e. a standard collection of strings. Now, we need to add a using statement at the top of the program file to the query implementation of NMF Expressions. Add the code from Listing 4 to the top of the program file.

```
using NMF.Expressions.Linq;
```

Listing 4. Registering the query implementation of NMF Expressions

As a consequence, the query implementation of NMF Expressions is used and thus, the variable `stateHubs` has the type `IEnumerableExpression<string>`. This adds a method to obtain an incrementalized version of the query through the `AsNotifiable` method.

[9] https://github.com/NMFCode/NMFDemo/blob/master/Example.fsm.

```
stateHubs.AsNotifiable().CollectionChanged += (o,e) => {
  if (e.NewItems != null)
      for (string name in e.NewItems) { Console.WriteLine("{0}␣is␣a␣new␣hub", name); }
    if (e.OldItems != null)
      for (string name in e.OldItems) { Console.WriteLine("{0}␣is␣no␣longer␣a␣hub", name); }
};
```

Listing 5. Adding handlers when the analysis results have changed

To verify the change propagation, visualize changes made to the state hub analysis through the code shown in Listing 5. Normally, the method `AsNotifiable` is a very expensive operation, thus one would save the return value.

```
1 var checkStock = fsm.States[1];
2 checkStock.Outgoing.Add(new Transition() {
3    Input = "items␣are␣for␣free",
4    Target = fsm.States[2]
5 });
```

Listing 6. Adding a new transition to imply a new hub

Add some change operations after registering the handler, step through the console application and see how new hubs are immediately shown in the console. For example, you can use the code listed in Listing 6 to create a new transition to skip payment when items of the order process are for free. As a consequence of this change, a message will pop up in the console that a new hub has been detected directly after Line 2–5 of Listing 6 have been executed.

A.5 Creating a Model Transformation

Now, we are going to transform the state machine model into a different model, for instance in a Petri net.

At first, we need to add the libraries to run model transformations in NTL. The easiest way to get them is to download them as another NuGet package. Install NMF Transformations through the NuGet command shown in Listing 7 or again through the GUI.

```
PM> Install-Package NMF-Transformations
```

Listing 7. Installing NMF Transformations

A model transformation in NMF Transformations is a special class, inheriting from `ReflectiveTransformation`. Thus, create a new class by adding a new class to the project. Download the model transformation `FSM2PN` from finite state machines to Petri nets from the examples page[10] and copy its contents into the new file.

[10] https://github.com/NMFCode/NMFDemo/blob/master/FSM2PN.cs.

To run this model transformation, we need to instantiate the model transformation, initialize it and run it. The initialization can be reused for multiple passes of a model transformation, in case the model transformation initialization is costly. To apply the model transformation, we need to pass the source and target model type as generic parameters. The transformation then selects an appropriate rule to start with and traverses the transformation through the rule dependencies. Thus, we can ask the transformation to transform states or entire state machines.

```
var transformation = new FSM2PN();
var context = new TransformationContext();
var petriNet = TransformationEngine.Transform<StateMachine, Net>(fsm, context);
```

Listing 8. Running NMF Model Transformations

After the transformation, the `context` object can be used for tracing purposes.

A.6 Saving Result Model to a File

In NMF, the serialization information of model elements is attached directly to the model representation classes. The NMF serializer uses this information and interprets how the model should be serialized to XMI. To serialize the Petri net, we simply save it into our model repository (or create a new one). Any referenced model element already contained in another existing file is referenced through a fully qualified reference.

```
repository.Save(petriNet, "Example.pn");
```

Listing 9. Serializing models in NMF

To save a model element to a file, it is sufficient to call the Save method on the repository such as shown in Listing 9. Verify that you can open this file in Eclipse.

References

1. Vaquero-Melchor, D., Palomares, J., Guerra, E., de Lara, J.: Active domainspecific languages: making every mobile user a modeller. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 75–82. IEEE (2017)
2. Hinkel, G., Happe, L.: Using component frameworks for model transformations by an internal DSL. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for Component-Based Software Systems Co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), ser. CEURWorkshop Proceedings, vol. 1281, CEUR-WS.org, pp. 6–15 (2014)
3. Hinkel, G., Happe, L.: An NMF solution to the TTC train benchmark case. In: Proceedings of the 8th Transformation Tool Contest, a Part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, ser. CEUR Workshop Proceedings, vol. 1524, CEUR-WS.org, pp. 142–146 (2015)

4. Hinkel, G.: An approach to maintainable model transformations using an internal DSL. Master's thesis, Karlsruhe Institute of Technology (2013)
5. Hinkel, G., Goldschmidt, T.: Tool support for model transformations: on solutions using internal languages. In: Modellierung 2016 (2016)
6. Wagelaar, D., Van Der Straeten, R., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. Softw. Syst. Model. **9**(3), 285–309 (2010)
7. Hinkel, G., Burger, E.: Change propagation and bidirectionality in internal transformation DSLs. Software & Systems Modeling (2017)

Author Index