Synthesis of Infinite-State Abstractions and Their Use for Software Validation

Carlo Ghezzi*, Andrea Mocci, and Mario Sangiorgio

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria,
P.za Leonardo Da Vinci 32, 20131 Milano (MI) Italy
{ghezzi,mocci,sangiorgio}@elet.polimi.it

Abstract. In the recent years, several research efforts have been devoted to developing approaches to synthesize specifications of software behavior. Most of the proposed approaches addressed the inference of finite-state abstractions. The synthesized abstractions have been integrated in different validation scenarios, such as testing. While finite-state models can be effectively used as models of a software component's behavior for certain specific purposes, they can hardly be used as full-fledged specifications. Because of their very limited expressive power, they cannot represent some of the component behaviors and may lead to synthesizing too coarse abstractions. In this paper, we survey a set of approaches that instead infer infinite-state abstractions, which can be used to express richer sets of behaviors of a software component in a black-box manner. For such approaches, we also discuss the few existing applications to software validation. In particular, we discuss the limitations and identify how, in principle, they can be used in different validation scenarios and how this opens new research directions.

1 Introduction

A formal specification is a description of the behaviors of a given software expressed in a certain mathematical notation with a clear semantics. Formal specifications are important and often essential for many validation activities. Examples of such activities are testing [1], where specifications can be used as oracles to check the correctness of an implementation for a certain set of inputs, or model checking [2], where specifications have both the role of models of software artifacts and properties to be checked on the model itself.

In practice, the cost of producing a component's specification is often as high as code writing, and thus producing the component itself. Moreover, a formal specification requires mathematical skills that may not be possessed by developers. These are among the reasons why specifications are often absent for real-world software components. When present, specifications are given through

* This research has been partially funded by the European Commission, Programme IDEAS-ERC, Pro ject 227977-SMScom.

S. Iida, J. Meseguer, and K. Ogata (Eds.): Futatsugi Festschrift, LNCS 8373, pp. 276–295, 2014.

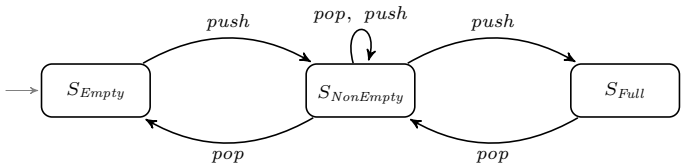


Fig. 1. A behavior model of a bounded stack

natural language in an informal way, that is not amenable to the automatic validation approaches described above. In addition, no guarantee can be assumed that the specification and the implementation are synchronized. Very often, they diverge because implementations are maintained without making the corresponding changes to the specification.

A recent branch of research activity in software engineering has been devoted to addressing the problems due to a missing specification by proposing the automatic synthesis through the analysis of existing software. The pioneering work described in DAIKON [3] goes exactly in this direction. Most of the work involving software specification synthesis has focused on finite-state abstractions of software behavior [4]. Finite-state abstractions may capture an important behavioral aspect of software components, that is, the protocol of interaction with the component. Intuitively, an interaction protocol describes the legal sequences of operations that are valid from the client's point of view when the client calls operations available through the component's interface.

Properties of the interaction protocol typically express precedence relations. For example, a component that represents a file requires that the file should be open before a write operation can be performed; that is, write can only be called after (a successful) open operation. Such properties are naturally expressible with an automaton, or in general with a finite-state abstraction that may not possess all the properties of an automaton. Semiautomata, that is, automata with no final states, are typically used to express interaction protocols, since the notion of a final state is not useful to express component behaviors.

Examples of finite-state models of software components are the ones inferred by ADABU [5,6], which uses dynamic analysis, and CONTRACTOR [7], which uses static analysis to derive behavior models from C programs. Figure 1 shows a behavior model of a bounded stack as inferred by CONTRACTOR; for example, the model describes the fact that the *pop* operation is not enabled in S_{Empty}, imposing a precedence relation on the legal sequences of operations on the component, that requires at least a *push* operation to be called before any call of *pop*.

Inferred behavior models have been used for many validation activities; examples include test case generation [8,6], integration in model checking [9], and runtime verification [10]. However, behavior models are formalisms that capture only a subset of the possible behaviors of the analyzed component. In particular, being finite-state machines, they must abstract away any collection-like behavior, like LIFO or FIFO behaviors, because these cannot be represented

with a finite-state abstraction. For this reason, finite-state machines are most of the times *models* of certain behaviors exhibited by a given software, rather than full-fledged specifications of it.

The motivation of this paper is twofold. First, we critically survey the field, focusing on techniques that infer specifications that instead consist of infinite-state abstractions, which potentially may achieve the role of full-fledged specifications of software components. For example, such abstractions are contracts [11] or algebraic specifications [12,13], which potentially can capture such infinite-state behaviors, like collection-like behaviors, that finite-state machines may represent only in a very imprecise way, yielding very coarse abstractions.

Finally, we are interested in exploring the potential usages of inferred infinite-state abstractions that may reveal new research directions. In fact, while several approaches to validation that use inferred behavior models have been studied and proposed, very few exist that use inferred infinite-state abstractions in similar scenarios. To this aim, we first critically analyze such existing works, and then we outline possible future work considering the existing literature where infinite-state abstractions are considered to be present, and where an inference step could be potentially integrated.

For the sake of clarity, hereafter we discuss some of the assumptions we make in this paper and we describe the main terms we use. First, we refer to software components that define *abstract data types* (ADT), implemented as *classes*. We assume that the class only exports *methods* through its *interface*. A method represents an operation that can be used to operate on instances of the ADT (also called objects). Client components can only use these exported operations to interact with a given component. We distinguish among the following kinds of operations:

Observers : These are operations that return to the client some information related to the state of the object upon which they are invoked. Observers may be *pure* or not. A pure observer can only observe and cannot modify the internal state of the object.

Modifiers : These are operations that change the state of the object they are applied upon. If a class exports modifiers, the instantiated objects are said to be *mutable*.

Terms, or *traces*, represent sequences of operations. While the two words are often used interchangeably in formal approaches to software specification, in the area of software testing and analysis usually a trace denotes an *execution trace*, that is, an executed sequence of operations of a given implemented software component. In this paper, for the sake of clarity, we will always refer to such notion as execution trace.

Structure of the Paper. The paper is organized as follows. Section 2 discusses the state of the art about synthesis of infinite-state abstractions from software components. We classify the existing approaches by the classes of infinite state abstractions that they can synthesize, like contracts and algebraic

Table 1. Works surveyed on the State of the Art

Approach	Ref.	Specification	Abstract	Input	Analysis
Daikon	[3,14]	Contract	No	Ex.Traces	Dynamic
DySy	[15]	Contract	No	Code + Ex.Traces	Dynamic + Static
Axiom Meister	[16]	Contract	Yes	Code	Static
AutoInfer	[17]	Contract	Yes	Tests	Dynamic
KindSpec	[18]	Contract	Yes	Code	Static
Heureka	[19]	Algebraic Spec	Yes	Tests	Dynamic
Adiheu	[20]	Algebraic Spec	Yes	Tests	Dynamic
Sabicu	[21]	Algebraic Spec	Yes	Tests	Dynamic
AbsSpec	[22]	Algebraic Spec	Yes	Code	Static
Spy	[23]	Intensional BM	Yes	Tests	Dynamic

specifications. Then, Section 3 identifies possible validation scenarios where such inferred infinite-state abstractions could be used, considering existing approaches and outlining promising research directions. Finally, Section 4 derives conclusions of this paper.

2 Synthesis of Infinite-State Abstractions: State of the Art

In this section, we will introduce the existing synthesis approaches that address infinite-state abstractions. Table 1 reports the main features of the surveyed approaches, classified mainly according to the class of infinite-state abstraction (specification) they synthesize. Moreover, we distinguish whether the specification is *abstract* (that is, expressed in terms of what a client can observe externally), what is the *source artifact* of the analysis, and what kind of analysis (static or dynamic) is used to infer the specification.

The section is structured according to the kind of specification synthesized by each of the surveyed approaches. We start by describing approaches that infer contracts (Section 2.1), then approaches that synthesize algebraic specifications (Section 2.2), and finally specifications based on trace assertions (Section 2.3).

2.1 Inferring Contracts

Contracts are a popular methodology to specify the behavior of software components in general, and they have been successfully applied to infinite-state components too. Basically, a *contract* [11,24] uses pre/post-conditions to specify the behavior of each operation in isolation. The pre-condition states what has to be true to invoke the operation (i.e., it states an obligation for the client); the post-condition states what has to be true when the operation terminates (i.e., it states an obligation for the object onto which the operation is applied).

There are five main approaches that implement inference of contracts for infinite-state components:

public class StackAr
private Object[] theArray;
private int topOfStack;
Precondition : capacity ≥ 0
Postcondition : capacity = this.theArray.length
this.topOfStack = -1
this.theArray[] elements = **null**
public StackAr(**int** capacity) {...}

Fig. 2. The contract inferred by DAIKON for the STACKAR constructor

- DAIKON, an invariant detector that can be used to infer contracts of data abstractions;
- DYSY, which integrates dynamic analysis and symbolic execution to infer contracts;
- AXIOM MEISTER, which infers contracts for modifiers, expressed in terms of observer return values, using symbolic execution;
- AUTOINFER, that infers abstract postconditions of modifiers for components written in the EIFFEL language;
- KINDSPEC, which infers pre- and post-condition like specifications of C programs using the MATCHING LOGIC semantic framework [25].

Daikon. The DAIKON [3,14] invariant detector has been a pioneering work in the area of specification inference. DAIKON analyses the values of program variables at specific program points as a result of test case invocations. Starting from the results of these test case invocations, DAIKON infers invariant properties that hold at the recorded program points. For example, it may infer that the value of variable x is always greater than or equal to 10 before a statement that contains a division by s. An invariant holding at the entry point of an operation represents its preconditions, while an invariant holding at exit points represents a postcondition.

The inferred invariants predicate about program variables, including internal fields of classes. Consider, for example, a reference implementation of a bounded stack in JAVA, called STACKAR; this ADT is typically implemented with an array and an integer value pointing to the top of the stack. Figure 2 shows an example of invariants inferred by DAIKON, representing likely preconditions and postconditions of a STACKAR constructor which initializes it with a specific capacity.

Daikon works by generating candidate invariants out of a rich grammar of patterns, and then checking if they hold at specific program points. Invariants are reported only if there is enough statistical evidence that they do not hold by chance, and several optimizations are performed to get better results in terms of performance and relevance of reported invariants. Such optimizations include, for example, suppression of weaker invariants, that is, invariants that are logically implied by other ones.

DySy. One of the main problems with Daikon and in general of dynamic invariant detection is that it is hard, sometimes, to know in advance what are the possible patterns of invariants to be detected, and so the approach could fail in deriving interesting behaviors of the component to be analyzed. To overcome this limitation, the DySy approach [15] integrates black box dynamic analysis with symbolic execution, which is a white box, static analysis technique. By using symbolic execution, DySy is able to derive operation pre- and post-conditions based on the actual code behavior; in this sense, DySy is able to infer a method specification without using invariant patterns. However, symbolic execution is unable to generalize code behavior in the presence of loops or recursive operations; in this case, the approach uses some ad-hoc heuristics to support common iterative structures.

In the case of both Daikon and DySy, there are some fundamental problems in applying specification synthesis approaches to extract contracts of data abstractions. The main problem is that the state must be expressed in function of some variables that represent the state of the component. In principle, the specification of a component must be *abstract*, that is, *implementation independent*. In other words, specifications should respect the *information hiding* [26] principle. It should be expressed only in terms of the operations that are visible at the component's interface. Although useful for many development activities like testing, the invariants extracted by Daikon and DySy are not abstract and they represent code behavior expressed in terms of the component internals.

Axiom Meister. This tool [16] infers contracts from the static analysis of .Net programs. The tool requires the developer to choose the modifier methods he wants to analyze and its related observers. The tool requires observers to be *observationally pure*, i.e. they are only allowed to change the state in a way that it is invisible to clients. Then, Axiom Meister produces an abstract description of the modifier behavior in terms of the values returned by observer methods. Figure 3 shows the specification inferred by Axiom Meister for the push operation of a bounded stack.

Axiom Meister's inference approach is based on symbolic execution of the modifier method under analysis, which tries to explore all the possible execution paths; in symbolic execution, for each path, there is a corresponding path condition stating the symbolic constraints required for its execution. In general, path conditions express constraints over the data structures used to implement the operation and its enclosing class. The tool aims at producing a specification for an ADT and thus it has to find an abstraction of path conditions relying only on class observers. This process produces many path specific axioms, which are finally merged and simplified to obtain the more compact and readable specification. The inferred specification can be either used by humans or by Spec#.

AutoInfer. The AutoInfer approach presented in [17] provides another interesting approach that partially infers contracts with dynamic analysis but without using component internals. In fact, AutoInfer expresses behavior in function of

```
void push(Object x)
    requires size()<capacity() otherwise FullStackException
    ensures top() = x
    ensures size() = old(size()) + 1
    ensures capacity() = old(capacity())
```

Fig. 3. The SPEC# contract inferred by AXIOM MEISTER for the PUSH method of a bounded stack

observer return values. The approach targets Eiffel, an object-oriented language that supports *design by contract* [11], a development methodology that focuses on specifying the behavior of software components through contracts. The inference approach proposed by the authors uses novel dynamic inference techniques to infer modifier postconditions. Intuitively, such postconditions are properties that express how observers change their returned values after the invocation of an operation[1].

In particular, the approach supports the inference of two peculiar kind of assertions in modifier postconditions:

- assertions that involve quantification, that are useful to express frame properties, like that all the elements of a collection are still present after an operation invocation;
- assertions that involve implications, useful to identify which conditions trigger a particular different behavior.

AUTOINFER is based on dynamic analysis like DAIKON; the test cases used as inference base are generated by using random testing. The approach uses a modified version of AUTOTEST [27], which prunes generated test cases when they satisfy the operation preconditions. The operation preconditions are essentially assumed to be (correctly) written by the developer.

The main limitation of the approach is that it just focuses on the inference of postconditions of modifiers; the authors tailored the proposed inference techniques to postconditions because a previous work identifies preconditions as normally well written in contract-based development approaches. For this reason, the approach is not easily extensible to preconditions.

KindSpec. The KINDSPEC approach presented in [18] uses a static analysis technique based on the K framework to infer specifications for KERNELC programs. In particular, the tool focuses on modeling the behavior of heap-manipulating code. For each modifier operation, the tool finds a specification in the form of a set of facts represented by logic implications. Figure 4 shows an example of a specification inferred by KINDSPEC. An important difference with respect to the specifications produced by other tools is that KINDSPEC only

[1] In the Eiffel jargon, the specific language targeted by the approach, observers are called queries and modifiers are called commands.

```
isnull(s) = 1                    ⟹ isnull(s') = 1
-------------------------------------------------------------------------
isnull(s) = 0              ∧ ⟹ top(s) = top(s')
size(s) = capacity(s)
-------------------------------------------------------------------------
isnull(s) = 0              ∧ ⟹ isnull(s') = 0 ∧ top(s') = x ∧ size(s') =
size(s) < capacity(s)          size(s) + 1
```

Fig. 4. The contract inferred by KindSpec for the `push(Stack s, Object x)` function for a bounded Stack

looks for invariants involving calls to observer functions and the return value of the analyzed modifier. It does not produce invariants containing predicates over the implementation details of a class. It is worth to say that, since it is not possible to guarantee that a KernelC observer is pure, each function call in the logic formulas is assumed to be evaluated independently from the others, under the same initial conditions, to avoid the need of making assumptions on possible side effects.

The inference algorithm implemented in KindSpec relies on the symbolic execution engine of the Matching Logic verifier MatchC. This choice makes it possible for KindSpec to statically analyze the source code instead of concrete execution traces. The framework also ensures the correctness of the inferred specifications. In fact, they can check the inferred specifications with the MatchC verifier. A peculiarity of this approach is that KindSpec does not reason about the whole heap. It instead separates the different parts of the heap and its algorithms can reason only about the relevant ones for a specific function.

KindSpec is interesting from a particular point of view, which serves as a bridge from contracts to algebraic specifications for our classification of inference approaches. It is important to emphasize that while the MatchC verifier – on which the programs are interpreted to infer specifications – is based on algebraic rewrite rules, the inferred specifications themselves are not presented and interpreted with an algebraic style. In fact, each inferred fact represents the pre- and the post-state of the component itself explicitly, in a style that is typical of contracts, justifying our classification. However, by using only observers, such facts could be easily rewritten, and possibly interpreted, as conditional axioms in a typical algebraic specification style.

2.2 Synthesis of Algebraic Specifications

When specifying components with contracts, the focus is on each operation in isolation, and some kind of model is used to predicate about the state of the component to be specified. This can be the value of internal variables, as in the case of Daikon, or the return value of observers, like in the case of AutoInfer. Both, however, have pitfalls. The former case leads to a violation of the information hiding principle, since the specification ends up referring to implementation details that should instead remain hidden. The latter instead assumes that enough observers are available to support the effect of any other operation. A different point of view is adopted in the case of algebraic specifications, where

$\forall x : Stack, c : Integer, e : Object \mid$	
$pop.state(push.state(x, e))$	$= x$
$size.retval(Stack(c))$	$= 0$
$size.retval(push.state(x, e))$	$= size.retval(x) + 1$
$top.retval(push.state(x, e))$	$= e$
$contains.retval(Stack(c), e)$	$=$ **false**
$contains.retval(push(x, e), e)$	$=$ **true**

Fig. 5. An algebraic specification of an unbounded Stack as inferred by HEUREKA

the focus is on the whole component to be specified, and the behavior of operations is specified implicitly (and not explicitly) through algebraic axioms.

Each axiom in an algebraic specification is an equation that prescribes when two different sequences of operations, for a certain state and for certain parameters, are *equal*. The interpretation of this equality depends on the semantics of the algebraic specifications; for example, the semantics may impose that the sequences of operations are equivalent in the sense that they will expose the same observable behavior to the component clients.

Several approaches have been investigated to synthesize specifications that fall in the area of algebraic specification. The main ones are surveyed below.

Heureka. HEUREKA [19] is the main existing approach that infers algebraic specifications. HEUREKA explicitly targets JAVA classes and uses dynamic analysis. Furthermore, HEUREKA uses a particular semantics for algebraic axioms, called *behavioral* or *hidden semantics* [28], which is based on the concept of *behavioral equivalence*. Intuitively, a behavioral equivalence relation clusters elements of the algebra which cannot be distinguished by any possible sequence of operations.

Figure 5 shows an algebraic specification of an unbounded stack as inferred by HEUREKA. Because the synthesis approach targets JAVA, each method is potentially modeled with two operations: i) an operation modeling how state is changed, if the method is not pure – the operation is denoted with the **.state** suffix; ii) an operation modeling the return value of the method, if this is not **void** – the operation is denoted with the **.retval** suffix. This is a typical choice when modeling classes of object-oriented languages with algebraic specifications.

The technique operates with a fully black-box approach; the input consists of a class' public interface and a set of actual values for method parameters, called *instance pool*. HEUREKA starts by generating terms, that is, sequences of operations, and then groups them by checking if they are behaviorally equivalent. Behavioral equivalence is in general undecidable, so the tool checks it up to a certain maximum depth of contexts. Once equivalent terms have been detected, the tools tries to generalize them by introducing universally quantified variables, obtaining candidate axioms. Axioms are then tested and reported if no counterexample is found.

HEUREKA has been evaluated against implementations of data abstractions mainly from the Java Development Kit (JDK).

Adiheu. Another work that infers algebraic specifications is ADIHEU [20]. The approach uses behavior models to improve the inference process of HEUREKA. Essentially, behavior models are used to reduce the checks needed to establish if two terms are behaviorally equivalent. In the case of ADIHEU, the finite-state abstraction of the component is used as an intermediate model that is easier to synthesize, but whose synthesis dramatically improves the performance of the algebraic specification synthesizer of HEUREKA. The approach reduces the needed number of method invocations for the component under analysis from 50% to almost one order of magnitude.

Sabicu. SABICU [21] proposed an approach to infer algebraic specification similar to HEUREKA, but supporting also the inference of conditional algebraic axioms. Compared to HEUREKA, it is less general in the sense that the structure of possible axiom is derived from predefined templates, not from the generalization of equations classified by behavioral equivalence. Such predefined patterns may be enriched with *conditional extensions*, that represent specific conditions for the axiom to be applied. For example, consider the *contains* method of the stack example; then, the following axiom can be inferred by SABICU:

$$\forall x : Stack, e, f : Object \mid \begin{array}{l} contains.retval(push.state(x, e), f) = \textbf{true if } e = f \\ \textbf{else } contains.retval(x, f) \end{array}$$

The axiom expresses the fact that if *contains* is called after a *push*, it returns true if the parameter value for both methods is the same; otherwise, it returns the return value of *contains* called on the rest of the stack.

Another important aspect of SABICU is that it keeps track of a statistical metric of axioms, that is, the number of instances that satisfy the axiom itself. Thus, it not only derives common properties, that is, properties that hold for every possible instance of an ADT, but also special axioms that hold only for a subset of the tested instances. This aspect is useful to derive axioms whose holding conditions are too complex and not supported by the inference patterns of SABICU, but that could be potentially derived manually.

AbsSpec. ABSSPEC [22] is a tool to automatically infer high level, property-oriented specifications in the form of algebraic equations for CURRY, a lazy functional logic programming language. These features of CURRY require a careful definition of program semantics. The nature of the language requires different equality relations to be defined in order to support features like free variables in formulas.

Like other algebraic specification recovery tools, ABSSPEC produces specifications in the form of sets of equations relating (nested) operation calls that have the same behavior. ABSSPEC is based on a white box static inference mechanism that is guaranteed to generate correct specifications. The inference technique is

```
data Stack a = S [a]

new      :: Stack a
isEmpty  :: Queue a -> Bool
push     :: a -> Stack a -> Stack a
pop      :: Stack a -> Stack a
top      :: Stack a -> a

-- Inferred algebraic specification
-- contextual equivalence
(pop (push x (pop y))) = (pop (pop (push x y)))
(top (push x (push y z))) = (top (push x (pop (push y z))))
(pop (push x (pop (pop y)))) = (pop (pop (pop (push x y))))
(pop (push x (pop (push y z)))) = (pop (pop (push y (push x z))))
(pop (push x (pop (push y z)))) = (pop (pop (push x (push y z))))
(top (push x1 (push x3 (push x2 x4)))) =
    (top (push x1 (push x2 (push x3 x4))))

-- computed result equivalence
(top (push x new)) = x
```

Fig. 6. A two-sided queue algebraic specification inferred by AbsSpec

based on an abstract semantics for the Curry language: a condensed goal-independent fix-point semantics that has been specifically designed to model the small-step behavior of rewriting [29] for logic functional programming languages.

The completeness of the inferred specifications depends on the analysis bounds in term of trace length and analyzed functions the user decides to set. The tool is guaranteed to infer correct and complete specifications within these bounds.

Figure 6 shows the inferred algebraic specification for a stack. For the example we selected to consider the `push`, `pop`, `top`, and `isEmpty` functions. The inferred algebraic specification includes two different kinds of logic formulas. The first set of equations uses the contextual equivalence, which checks whether two terms are equal within any context. The last formula instead uses the *computed result* equivalence relation, in which all the possible outcomes for the left side equals to the results for the right side. The latter is the usual equality relation for functional languages.

2.3 A Synthesis Approach Based on Trace Assertions

Algebraic specifications are useful to infer some interesting properties of operation interaction (like idempotent and equivalent traces), but in some cases it is hard to use them as a specification language. In some cases, algebraic specifications require *hidden functions*, that is, operations that are used only for specification purposes and that are not exposed to the clients to be accessible.

For many reasons, the use of hidden functions has been criticized as a problem with respect to information hiding, and some authors have considered this necessity of algebraic specifications as a violation of the principle (see for example [30] and also some early work on software specification [31]). In fact, they may convey design and implementation decisions.

Obviously, this problem also hinders the capability of algebraic specifications to be inferred in the case of components that would require hidden functions.

Canonical Traces : $Stack(c).push^N(d_i) \mid N \leq c$

operation	Pattern	Equivalence
$t.pop()$	$t = s.push(d)$	s
$t.push(e)$	$t = Stack(c).push^c(d_i)$	t
$t.capacity() : c$	$t = Stack(c).s$	t
$t.size() : 0$	$t = Stack(c)$	t
$t.size() : k$	$t = Stack(c).push^k(d_i)$	t
$t.top() : d$	$t = s.push(d)$	t
$t.contains(e)$: **true**	$t = s.push(e).q$	t
$t.contains(e)$: **false**	$t = Stack(c).push^c(d_i) \mid d_i \neq e$	t

Fig. 7. A TAM specification of a bounded stack

Hidden functions, in general, encapsulate an abstract state that depends on the component behavior itself.

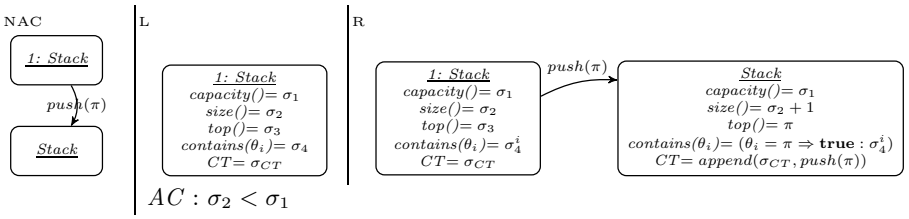
The trace assertion method (TAM) [32,30] is a specification formalism and notation introduced to deal with the problems of information hiding violation in algebraic specifications. A particular set of traces, called *canonical traces*, is chosen to uniquely identify the state of the component, and assertions (that is, predicates) on canonical traces are used to the behavior of operations.

In TAM, an operation is specified in a tabular notation which maps a given pattern in the trace to the behavior of the operation. For this reason, equations describing the behavior of operations are explicit, in the sense that they explicitly describe their behavior in terms of the canonical trace model.

Figure 7 shows the TAM specification of a stack with a tabular notation. By allowing arbitrary predicates to specify the structure of traces, TAM solves the problem of hidden functions.

We are not aware of published work that directly targets trace assertions as a specification formalism for synthesized specifications. We have instead developed the SPY approach [23], which uses a closely related method to model the state of an infinite-state data abstraction.

Spy. SPY [23] is a specification synthesis method that targets JAVA classes like HEUREKA, but it uses a different class of target specifications. SPY synthesizes so-called *intensional behavior models*. As we discussed briefly in the introduction, a behavior model is a finite-state abstraction where each transition represents a modifier invocation and the states describe, with a certain level of abstraction, observer return value in a particular state. An intensional behavior model is a generalization of a particular kind of behavior model, called *behavioral equivalence model (*BEM*)*, where each state represents a class of behaviorally equivalent objects, and it represents a subset of all the possible behaviors of the component (e.g., for a subset of possible parameters for methods, or for terms up to a certain length). An intensional behavior model generalizes a BEM by using an



(a) Push Rule

Fig. 8. An intensional behavior model rule describing the behavior of a *push* operation in a non-full bounded Stack

generative approach, similar to a graph grammar: each operation is described by a rule which describes how new states, and new transitions, can be added to an existing BEM to explain new behaviors. The generalization is possible because each state is uniquely identified by a canonical trace, as in TAM.

Figure 8 shows a rule describing the behavior of the *push* operation of a stack. Essentially, it describes how a new state on a BEM can be generated after the invocation of a push operation. First, the rule describes how the observer return values change – that is, *size* is incremented by 1 and *top* returns the inserted element; second, the rule describes the value of the new canonical trace for the state by appending the last operation to the previous canonical trace.

SPY first infers a BEM from the dynamic analysis of a JAVA class, as in HEUREKA. Essentially, it has three steps:

- for a user-specified instance pool, it generates terms up to a certain length and classifies them by behavioral equivalence, generating a BEM;
- then, SPY uses heuristics based on a metric for object distance to identify a candidate set of canonical traces for an inferred BEM;
- finally, a generalization step uses invariant detection (à la DAIKON) to detect both trace assertions describing the behavior of modifiers and other invariants describing the behavior of observers; such invariants are used to generalize BEMs to intensional behavior models.

The use of trace assertions, essentially, makes SPY able to infer specifications where methods like HEUREKA would explicitly require hidden functions.

3 Use of Inferred Specifications for Validation

In the introduction of this paper we discussed how inferred specifications may play a role beyond specification recovery and documentation improvement. We mentioned, in particular, their use in software validation. Hereafter we first survey and discuss existing work that uses inferred infinite-state models for validation purposes (Section 3.1). Then we outline some possible research directions (Section 3.2).

3.1 Existing Approaches

Hereafter, we explore the related work that uses the results of inference of infinite-state abstractions to perform different kinds of validation – either of the inferred specifications themselves or of the artifacts under analysis. Existing work in this area is reviewed below for the classes of specification we identified in this paper.

Contracts. Most existing validation approaches that use inferred infinite-state specifications involve contracts. A number of such techniques embed contract inference in more complex workflow that includes static analysis and testing. The use of Daikon, in this context, has been pretty intense [33]. A full survey of all the usages of Daikon is outside the scope of this paper, mainly because the contracts inferred by Daikon are in general not abstract, that is, they normally express pre- and post-conditions in term of a class' internal representation.

A notable example is provided by DSD-Crasher [34], which combines dynamic contract inference, static code analysis, and testing. The first phase of the approach relies on Daikon for specification inference. In this context, the inferred specification acts as an oracle of the intended program behavior. The next phases of DSD-Crasher focus on detection of possible bugs, which is performed in two separate steps. First, static analysis is used to find counterexamples that possibly violate the inferred contracts. Second, testing is performed to confirm that the issues found with static analysis are actually bugs. Both steps are important because static analysis may return an over-approximated set of issues containing some bugs that are not reachable in real executions.

A research approach that uses abstract contract inference for bug finding is Stateful Testing [35], which is based upon AutoInfer-synthesized Eiffel contracts. By using Stateful Testing it is possible to produce a suitable test suite that is both able to uncover bugs in the code and that can lead to the inference of more accurate contracts. The approach starts from the AutoTest random test case generation; the initial test suite is then improved by leveraging the contracts inferred with AutoInfer. The underlying hypothesis of the approach is that the initial test suite, which is generated randomly, likely misses to cover some behavior of the component under test. Thus, it is desirable to enrich such initial test suite to cover more behaviors and obtain a more complete test suite. Inferred contracts indeed provide an effective way to enhance a test suite. Stateful Testing leverages the information encoded in the inferred contracts to find test cases that violate their preconditions and postconditions. Adding these new test cases improves the number of faults that it can expose and thus, in the end, likely leads to the inference of more precise contracts.

Algebraic Specifications. Inference approaches like Heureka [19], [36], or Spy [23] interleave automated test generation with specification inference. For example, Heureka uses test generation also after algebraic axiom generation to prune invalid axioms generated by the generalization phase. In [36] the authors

propose an iterative approach where testing and specification inference are mutually enhanced by each other. Essentially, the approach starts from an initial set of tests that guide specification inference. Then such specifications are used to guide the automatic generation of new tests. In the initial iterations, such approach is very likely to generate test cases that violate the previously inferred specifications. The violating tests might be exercising either new program behaviors or exposing some fault that was not exercised in initial iterations of the approach. The technique has been implemented by considering both contracts as inferred by DAIKON, and algebraic specifications inferred with a mechanism that is closely related to the one of SABICU.

Through testing, it is also possible to provide an empirical comparison of the quality of inferred specifications by different methods. For example, the experimental validation of SPY against HEUREKA [23] has been performed by generating test cases and using the component under analysis as an oracle against the prediction of both inferred specifications, to compute the numbers of correct, wrong, or undetermined predictions.

Comparing inferred specifications is also the subject of [37]. This paper proposes a technique to automatically check the mutual consistency of two different infinite-state abstractions, that is, algebraic specifications and intensional behavior models of the same software component. The approach reduces the consistency problem to model checking. The evaluation considers (and simulates) typical situations that may arise in the context of specification inference. For example, different inference bases may get different (and inconsistent) behavior predicted by the two different inferred specifications; the approach is able to derive such inconsistencies that may potentially arise in the context of specification inference.

3.2 Potential Future Research Directions

The works analyzed so far demonstrate that specification inference can be the basis for interesting validation activities. Hereafter, we briefly analyze potentially interesting research directions that can be further. To this aim, we briefly discuss examples of existing literature that use non-inferred infinite state abstractions, and we discuss how inferred specifications could play a role. We explore two areas of possible applications: testing and debugging, and validation in the so-called *open world*.

Testing and Debugging. Testing and debugging techniques can get significant improvements in their effectiveness when accurate specifications are available. For example, specifications may act as oracles of the intended behavior of a piece of code. Testing tools can rely on this information to discriminate between the expected and actual behavior of the code under analysis.

Many existing works use contracts to enable the automatic generation of test cases that try to find input values violating them. For example, EIFFEL natively supports *design by contract*, and the research community developed several techniques to leverage such specification for automatic test case generation. These

approaches rely on contracts as oracles to drive the search performed by random testing techniques [27] or evolutionary algorithms [38]. These approaches explore the input spaces trying to find test cases that violate the precondition of a method, or that satisfy the precondition but violate the postcondition.

ASTOOT [39] uses algebraic specifications and term rewriting to automatically generate test cases for object-oriented software. The tool produces test cases to ensure the equivalence of all the sequence of operations that should bring an object in a given abstract state. It generates different sequences of operations and the assertions on the value returned by observer methods needed to check that the object is in the right abstract state. The presence of specifications is also useful to optimize an existing test suite. ROSTRA [40] relies on algebraic specifications to minimize the number of test cases contained in a test suite. Minimization is performed by eliminating redundant test cases, i.e. the test cases that cover the same abstract states, and removing the ones dealing with equivalent objects.

The aforementioned approaches work in the presence of an existing specification. Since very often in practice specifications are missing, incomplete, or unreliable, inference techniques can help to fill this gap. However, existing methods that apply to human-produced specifications cannot be simply transferred as they are to inferred specifications. The latter, in fact, reflect the actual behavior of observed code, plus generalizations; they do not express, per se, the *intended* program behavior. To enable application of approaches like the ones we reviewed above, one should first inspect the extracted specifications to ensure that they actually reflect the intended component behavior. This inspection can be non-trivial for infinite-state abstractions.

A different approach views specification inference and validation as integrated steps that mutually influence each other. In fact, it is also possible to envision a feedback loop in which specifications are built starting from existing test cases and, at the same time, they are used to construct new relevant test cases, reaching a point where the likelihood of finding new mismatching test cases from the implementation and the specification is relatively low. This approach has been explored by STATEFUL TESTING and can be traced back to the pioneering work on finite-state abstraction learning and testing approach called L^* [41]. If one applies such a technique to a reference implementation, then the specification is likely to capture the intended behavior and can be used, for example, for precise regression testing or even for program verification.

Service-Oriented Architectures and Open-World Software. Inference of infinite-state abstractions may play a relevant role in the context of modern software architectures, like service-oriented architectures, living in the so-called open world [42]. Such applications, in fact, are composed by using third-party components or services on which the developer has no control. Composition may even occur dynamically, at run time. In such contexts, the value of an inferred specification is particularly critical, since a service client has no possibility to inspect the implementation of a service.

The work by Bianculli et. al. [43] uses infinite-state abstractions in the context of a service-oriented environment. Specifically, algebraic specifications are used

to monitor the behavior of a stateful service, like a typical shopping cart used by most e-commerce Web applications. The proposed approach uses aspect-oriented programming [44] to monitor a BPEL service and the operations that are invoked on it. An algebraic specification interpreter (either the HEUREKA [45] or CAFEOBJ [46] one) is used to evaluate the monitored terms and check it against the observed operation result. In this work, the specification is assumed to be provided. However, this assumption may be irrealistic, considering the current state of the practice in the specification of service-oriented applications. Although a specification (both for functional and non-functional aspects) should in principle be available to support *service-level agreement* between service providers and users, in practice descriptions are informal and compliance with their evolving implementation may not be ensured.

Existing inference techniques may not be straightforwardly applied in this context. For example, the techniques for algebraic specifications, like HEUREKA, are computationally expensive and the number of method invocations required to infer a specification is relatively high. If these method invocations have to be applied to an existing service, they would negatively affect non-functional properties of the service to be analyzed. Furthermore, it can be unreasonable to expect each client should perform analysis on the same exposed service to obtain the specification. Instead, one should perhaps envision specific discovery services and mechanisms to perform specification inference. In addition, inference should apply to both functional and non-functional interface properties.

4 Conclusions

Infinite-state abstractions can provide very precise descriptions of behaviors that finite-state machines would instead ignore. However, it is hard to produce them and keep them consistent with implementations. Thus they are seldom used in the practice of software development. In this paper, we surveyed the research literature on specification inference for infinite-state abstractions, focusing in particular on two existing classes: pre-/post-condition based contracts and algebraic specifications. Furthermore, we outlined interesting validation scenarios where an inference step can play an important role, extending the applicability of existing work. Although very promising initial work has been focusing on the interplay between specification inference and validation, more research is needed to make it applicable and to pave the way for use of infinite-state abstractions in the practice of software engineering.

References

1. Young, M., Pezze, M.: Software Testing and Analysis: Process, Principles and Techniques. John Wiley & Sons (2007)
2. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation 98(2) (1992)
3. Ernst, M.D.: Dynamically Discovering Likely Program Invariants. Ph.D. thesis, University of Washington, Seattle, Washington (August 2000)

4. Robillard, M., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T.: Automated api property inference techniques. IEEE Transactions on Software Engineering 39(5), 613–637 (2013)
5. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with adabu. In: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, WODA 2006, pp. 17–24. ACM, New York (2006)
6. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA 2010, pp. 85–96. ACM, New York (2010)
7. De Caso, G., Braberman, V., Garbervetsky, D., Uchitel, S.: Program abstractions for behaviour validation. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 381–390 (2011)
8. Xie, T., Notkin, D.: Tool-assisted unit-test generation and selection based on operational abstractions. Automated Software Engineering 13(3), 345–371 (2006)
9. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX). FORTE XII / PSTV XIX 1999, pp. 225–240. Kluwer, The Netherlands (1999)
10. Mocci, A., Sangiorgio, M.: Detecting component changes at run time with behavior models. Computing 95(3), 191–221 (2013)
11. Meyer, B.: Applying "Design by Contract". IEEE Computer 25(10), 40–51 (1992)
12. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Informatica 10, 27–52 (1978), http://dx.doi.org/10.1007/BF00260922
13. Goguen, J., Malcolm, G.: Algebraic Semantics of Imperative Programs. Foundations of Computing Series. Mit Press (1996)
14. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1-3), 35–45 (2007); Special issue on Experimental Software and Toolkits
15. Csallner, C., Tillmann, N., Smaragdakis, Y.: DySy: Dynamic symbolic execution for invariant inference. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 281–290. ACM, New York (2008)
16. Tillmann, N., Chen, F., Schulte, W.: Discovering likely method specifications. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 717–736. Springer, Heidelberg (2006)
17. Wei, Y., Furia, C.A., Kazmin, N., Meyer, B.: Inferring better contracts. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 191–200. ACM, New York (2011)
18. Alpuente, M., Feliú, M.A., Villanueva, A.: Automatic inference of specifications using matching logic. In: Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, pp. 127–136. ACM, New York (2013)
19. Henkel, J., Reichenbach, C., Diwan, A.: Discovering documentation for Java container classes. IEEE Trans. Software Eng. 33(8), 526–543 (2007)
20. Ghezzi, C., Mocci, A., Monga, M.: Efficient recovery of algebraic specifications for stateful components. In: Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting, IWPSE 2007, pp. 98–105. ACM, New York (2007)

21. Xie, T., Notkin, D.: Automatically identifying special and common unit tests for object-oriented programs. In: Proc. 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005), pp. 277–287 (November 2005)
22. Bacci, G., Comini, M., Feliú, M.A., Villanueva, A.: Automatic synthesis of specifications for first order Curry programs. In: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming, PPDP 2012, pp. 25–34. ACM, New York (2012)
23. Ghezzi, C., Mocci, A., Monga, M.: Synthesizing intensional behavior models by graph transformation. In: IEEE 31st International Conference on Software Engineering, ICSE 2009, pp. 430–440. IEEE (2009)
24. Meyer, B.: Design by Contract: The Eiffel Method. In: International Conference on Technology of Object-Oriented Languages, p. 446 (1998)
25. Roşu, G., Ştefănescu, A.: Matching Logic: A New Program Verification Approach (NIER Track). In: ICSE 2011: Proceedings of the 30th International Conference on Software Engineering, pp. 868–871. ACM (2011)
26. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 1053–1058 (1972)
27. Meyer, B., Fiva, A., Ciupa, I., Leitner, A., Wei, Y., Stapf, E.: Programs that test themselves. Computer 42(9), 46–55 (2009)
28. Goguen, J., Malcolm, G.: A hidden agenda. Theoretical Computer Science 245(1), 55–101 (2000)
29. Comini, M., Torella, L.: A condensed goal-independent fixpoint semantics modeling the small-step behavior of rewriting. In: Kovacs, L., Kutsia, T. (eds.) SCSS 2013. EPiC Series, vol. 15, pp. 31–49. EasyChair (2013)
30. Janicki, R., Sekerinski, E.: Foundations of the trace assertion method of module interface specification, vol. 27, pp. 577–598. IEEE Press, Piscataway (2001)
31. Bartussek, W., Parnas, D.L.: Using assertions about traces to write abstract specifications for software modules. In: Bracchi, G., Lockemann, P. (eds.) Information Systems Methodology. LNCS, vol. 65, pp. 211–236. Springer, Heidelberg (1978)
32. Parnas, D.L.: A technique for software module specification with examples. Commun. ACM 15(5), 330–336 (1972)
33. Ernst, M.: Daikon-related invariant detection publications (2013), `http://groups.csail.mit.edu/pag/daikon/pubs/#daikon-methodology`
34. Csallner, C., Smaragdakis, Y., Xie, T.: DSD-Crasher: A hybrid analysis tool for bug finding. ACM Trans. Softw. Eng. Methodol. 17(2), 8:1–8:37 (2008)
35. Wei, Y., Roth, H., Furia, C., Pei, Y., Horton, A., Steindorfer, M., Nordio, M., Meyer, B.: Stateful testing: Finding more errors in code and contracts. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 440–443 (2011)
36. Xie, T., Notkin, D.: Mutually enhancing test generation and specification inference. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 60–69. Springer, Heidelberg (2004)
37. Ghezzi, C., Mocci, A., Salvaneschi, G.: Automatic cross validation of multiple specifications: A case study. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 233–247. Springer, Heidelberg (2010)
38. Silva, L.S., Wei, Y., Meyer, B., Oriol, M.: Evotec: Evolving the best testing strategy for contract-equipped programs. In: APSEC, pp. 290–297 (2011)
39. Doong, R.K., Frankl, P.G.: The astoot approach to testing object-oriented programs. ACM Transactions on Software Engineering and Methodology 3(2), 101–130 (1994)

40. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 196–205 (2004)
41. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
42. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issues and challenges. IEEE Computer 39(10), 36–43 (2006)
43. Bianculli, D., Ghezzi, C.: Monitoring conversational web services. In: 2nd International Workshop on Service Oriented Software Engineering: in Conjunction with the 6th ESEC/FSE Joint Meeting, IW-SOSWE 2007, pp. 15–21. ACM, New York (2007)
44. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
45. Henkel, J., Reichenbach, C., Diwan, A.: Developing and debugging algebraic specifications for java classes. ACM Trans. Softw. Eng. Methodol. 17(3), 14:1–14:37 (2008)
46. Diaconescu, R., Futatsugi, K., Iida, S.: Component-based algebraic specification and verification in CafeOBJ. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1644–1663. Springer, Heidelberg (1999)