Program Abstractions for Behaviour Validation*

Guido de Caso* Víctor Braberman*

* Departamento de Computación, FCEyN, UBA
Buenos Aires, Argentina
{gdecaso, vbraber, diegog}@dc.uba.ar

Diego Garbervetsky* Sebastián Uchitel*†

† Department of Computing, Imperial College
London, UK
s.uchitel@doc.ic.ac.uk

ABSTRACT

Code artefacts that have non-trivial requirements with respect to the ordering in which their methods or procedures ought to be called are common and appear, for instance, in the form of API implementations and objects. This work addresses the problem of validating if API implementations provide their intended behaviour when descriptions of this behaviour are informal, partial or non-existent. The proposed addresses this problem by generating abstract behaviour models which resemble typestates. These models are statically computed and encode all admisible sequences of method calls. The level of abstraction at which such models are constructed has shown to be useful for validating code artefacts and identifying findings which led to the discovery of bugs, adjustment of the requirements expected by the engineer to the requirements implicit in the code, and the improvement of available documentation.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*

General Terms

Behaviour model synthesis, automated abstraction, source code validation

1. INTRODUCTION

Code artefacts that have non-trivial requirements with respect to the order in which their methods or procedures ought to be called are commonplace. Such is the case for many API implementations and objects. In practice, descriptions of intended behaviour are incomplete and informal, if documented at all, hindering verification and validation of the code artefacts themselves and the client code that uses the artefacts. Hence, researchers have not relied on these descriptions and developed techniques to support the mining or synthesis of typestates [16] from API implementations which are then used to verify if client code conforms to the implemented protocol [1, 3]. Such approaches, however, address only part of the problem: they assume the code from which the typestate is extracted is correct; that it conforms to the ordering of methods or procedures intended at the time of design or developing the requirements for the API.

This work addresses the complementary problem of validating if API implementations provide their intended behaviour when descriptions of this behaviour are informal, partial or non-existent. Validation of API implementation behaviour can result in the identification of bugs in the code which induce undesired requirements, adjustment of the requirements expected by the engineer to the requirements implicit in the code, and the improvement of available documentation for that code.

In this work, we argue that an automatically constructed abstraction of an API implementation can be useful for validation against poorly documented requirements or the engineer's mental model and can lead to the identification of problems in the code, in the requirements or the engineer's understanding of both. Given that validation is an activity that requires human intervention, the level at which an API implementation is abstracted is key and has different requirements than those abstractions proposed for verification [17].

In this paper we present a novel technique for automatically constructing abstractions in the form of behaviour models from code artefacts equipped with requires clauses for methods. These models, similarly to typestates, encode all admissible sequences of method calls. The level of abstraction at which such models are constructed aims at preserving enabledness of sets of operations, resulting in a finite model with intuitive and formal traceability links to the code. This level of abstraction and the traceability links have shown to be useful for validation code artefacts and identifying findings that relate to bugs in code and problems in expected or documented requirements.

Literature on typestate synthesis refers to safety and permissiveness as the way to characterize abstraction properties: a typestate is *safe* [1] if no call sequence violates the library's internal invariants; the interface is *permissive* if it contains every such sequence. Previous approaches have aimed (e.g., [11]) at modular program analysis using types-

*This is a companion report extending sections 4 and 5 of the paper entitled "Program Abstractions for Behaviour Validation" submitted ICSE 2011.

ICSE 2011 Waikiki, Honolulu, Hawaii, USA

tates which are both safe and permissive for cases in which the library's internal state is finite, but may not be permissive for the infinite case. Our approach deals with infinite internal state space and is permissive at the cost of safety. We believe, and experience so far indicates, that this supports well identification of implementation and requirements issues.

The rest of this paper is organised as follows. We begin with an overview of the approach using a simple example (Section 2) and then provide a formal framework for our approach (Section 3). Subsequently, we present an algorithm for constructing enabledness abstractions (Section 4) and report on its use on a number of relevant source code subjects. (Section 5). Finally, we discuss related work (Section 6), ideas for future work and conclusions (Section 7).

2. OVERVIEW

In this section we provide a black box overview of the approach using a small example.

```
1  typedef struct node
2    {int data; struct node *next;} node;
3  typedef struct list {int size; node* first;} list;
4
5  list* l;
6
7  int inv() {return l==NULL || l->size >= 0;}
8  int List() {
9    l = (list*) malloc(sizeof(list));
10   if (l == NULL) return 0;
11   l->size = 0; l->first = NULL;
12   return 1;
13 }
14
15 int add_req() {return l!=NULL;}
16 int add(int data) {
17   node *tmp = l->first;
18   while (tmp->next != l->first)
19     tmp = tmp->next;
20   tmp->next = (node*) malloc(sizeof(node));
21   if(tmp->next == NULL) {
22     l = NULL; return 0;
23   }
24   tmp->next->data = data;
25   tmp->next->next = l->first;
26   l->size++; return 1;
27 }
28
29 int remove_req() {return l!=NULL && l->size > 0;}
30 void remove(){
31   node* new_first = l->first->next;
32   free(l->first);
33   l->first = new_first;
34 }
35
36 int destroy_req() {return l!=NULL;}
37 void destroy() {
38   // ...
39 }
```

Figure 1: A singly-linked list C implementation

Consider the C source code of Figure 1, which implements a singly-linked list. It features a `node` structure, which contains a `data` field and a pointer to the `next` node in the list (or to the first one, if standing on the last node). The list itself is stored in another structure, which holds the total number of elements and a pointer to the first node.

The implementation provides an initialization operation, which creates the `list` structure; an `add` operation which stores a new integer at the end of the list; a `remove` operation which eliminates the first element (if any) and a `destroy` operation which frees the memory used by the list and all its nodes. Note that besides its basic functionality, this list implementation is augmented with an invariant (`inv()`) and a requires clause for each of its operations (`add_req()`, `remove_req()`, and `destroy_req()`).

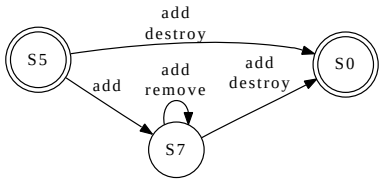

Figure 2: Singly-linked list enabledness abstraction

How can we validate if this list implementation provides the intended functionality when there is no such a formal and validated model of the intended functionality?

We propose is to automatically extract a behaviour model such as the one shown in Figure 2. In this model the concrete state space of the singly-linked list has been abstracted based on the set of operations the concrete states enable, that is, the set of operations for which their requires clauses hold. States that allow execution of `add` and `destroy` are represented by the abstract state `S5`, states that allow `add`, `remove`, and `destroy` are grouped into abstract state `S7`, and all concrete states that do not allow any operation are abstracted into `S0`. Note that *initial states* are marked with a *double circle*.

The behaviour model in Figure 2 allows an engineer to validate the implementation of the singly-linked list against his or her mental model of the intended behaviour of the source code. It is simple to see that model describes states which relate to whether a singly-linked list is empty (`S5`), non-empty (`S7`), or inactive (`S0`).

Consider now the `remove` operation. It is only featured in a transition that loops over state `S7`. This is suspicious, since it is indicating that whenever we erase an element from a non-empty list, we always end up having a non-empty list. There would seem to be a `remove` transition missing from `S7` to `S5`, which would model the case when the last element is removed from the list.

The implementation of `remove` does not ever empty the list. Surely, this is an unintended fault. Upon inspection of operation `remove` in Figure 1 we can observe that the list `size` field is not being decremented. Fixing this fault is straightforward and yields an enabledness behaviour abstraction that is the same as Figure 2, but with the addition of the missing `remove` transition from `S7` to `S5`.

The abstraction in Figure 2 could also prompt the discovery of interesting aspects of the implementation under analysis. For instance, both initializing the list and adding an element can lead to the terminal state `S0`. Inspection of the source code shows that memory availability has an impact on the list's behaviour. It is interesting to note that such observations, elicited easily from the abstraction, would require explicit modelling and or manipulation of the memory management aspects of the program's environment to be detected in verification-based approaches.

In summary, the example above illustrates how the depiction of an abstract model that integrates the behaviour of multiple procedures that use a common data structure for providing more complex services can support validation and aid identification of potential problems the implementation may have. The level of abstraction not only yields a compact

finite abstract model from an infinite concrete state space, but also allows allows tracing back concerns to the source code for identifying and fixing problems in the latter.

In the next two sections we show how enabledness-preserving abstractions like the one in Figure 2 can be built automatically from software implementations such as the one depicted in Figure 1.

3. FORMAL MODEL

In the following, we will use $\text{Pred}(D)$ to refer to the set of predicates over an arbitrary set D, formally defined as $\text{Pred}(D) \stackrel{\text{def}}{=} \{f \mid f : D \to \{\text{true}, \text{false}\}\}$. Additionally, $\mathbb{C}$ will denote the set of configurations to which programs evolve.

As we mentioned before, the object under analysis for our technique is the source code of a program. We will first define an *action system* as the semantic interpretation of a program's source code. This action system references the functions that act as requires clauses for each action, as well as the invariant and the initial condition. Then, we define the *semantics of an action system* as an infinite labelled transition system that captures its state space.

Definition 1 (Action System). *An* action system *over a set of configurations $\mathbb{C}$ is a structure of the form $AS_{\mathbb{C}} = \langle A,\ F,\ R,\ inv,\ init \rangle$, where:*

- $A = \{a_1, \ldots, a_n\}$ *is a finite set of* action labels.
- F *is an A-indexed set of* functions. *For each action label a, the function $f_a : \mathbb{C} \times \mathbb{Z} \to (\mathbb{C} \cup \bot)$ takes a configuration and an integer parameter*[1] *and has two possible outcomes:* i) *either it transforms the configuration, or* ii) *it does not terminate (represented by the $\bot$ symbol).*
- R *is an A-indexed set of* requires clauses. *For each action label a, the requires clause $R_a \in Pred(\mathbb{C} \times \mathbb{Z})$ is* true *when the action a is enabled for the given configuration and parameter.*
- $inv \in Pred(\mathbb{C})$ *is the* invariant.
- $init \in Pred(\mathbb{C})$ *is the* initial condition, *which indicates the starting configurations for the system.*

Example 1 (List). *A possible action system for the functional model of the list C implementation of the previous section is $AS_{\mathbb{C}} = \langle A,\ F,\ R,\ inv,\ init \rangle$ where:*

- $\mathbb{C}$ *is the C language memory model.*
- $A = \{add, rem, destroy\}$. *This set of actions indicates the* public interface *of the list implementation.*
- $F = \left\{ f_{remove},\ f_{add},\ f_{destroy} \right\}$
  Where these functions are the semantic interpretation of the corresponding C functions in Figure 1.
- R_{add} *yields* true *only for configurations on which the* `l` *variable is not* `NULL`.
- R_{rem} *yields* true *only for configurations that have a non-null* `l` *variable whose* `size` *field is positive.*
- $R_{destroy}$ *is the same as* R_{add}.
- *inv returns* true *only if* i) *the configuration has a* `NULL` `l` *variable; or* ii) *if* `l` *has a non-negative* `size` *field.*
- *init yields* true *only for configurations that have the* `l` *variable pointing* `NULL` *or to a structure such that:* i) *its* `size` *field is* 0 *and* ii) *its* `first` *field is* `NULL`. *This is the condition after applying the* `List` *function, which serves as constructor.*

Note that this example of an action system is rather arbitrary. For instance, we could have decided to leave the `destroy` *operation out of the labels set, which would have characterised a system with a smaller state space.*

We will use $\text{CodeOf}[f]$ to refer to the source code that is originally found in the program under analysis. For instance, consider the requires clause for the add operation presented in Figure 1. Its code is represented by $\text{CodeOf}[R_{\text{add}}]$ = `return head != NULL;`. Similarly, $\text{CodeOf}[f_{\text{add}}]$ is the fragment of lines 30–45 in Figure 1.

We now proceed to characterise the state space of an action system as an infinite deterministic Labelled Transition System (LTS). We define an LTS L as a tuple $\langle \mathbb{A}, \mathbb{S}, \mathbb{S}_0, \Delta \rangle$ where $\mathbb{A}$ is the action alphabet, $\mathbb{S}$ is a set of states, $\mathbb{S}_0 \subseteq \mathbb{S}$ is the set of initial states and $\Delta : \mathbb{S} \times \mathbb{A} \to \mathbb{S}$ is the partial transition function.

Definition 2 (Action System Semantics). *Given an action system $AS_{\mathbb{C}} = \langle A,\ F,\ R,\ inv,\ init \rangle$ we say that its* semantics *is given by an LTS $L = \langle \mathbb{A},\ \mathbb{S},\ \mathbb{S}_0,\ \Delta \rangle$ satisfying that $\mathbb{A} = A \times \mathbb{Z}$, $\mathbb{S} = \{c \in \mathbb{C} \mid inv(c) = \text{true}\}$ and $\mathbb{S}_0 = \{c \in \mathbb{S} \mid init(c) = \text{true}\}$. Also, for each $c \in \mathbb{S}$ and for each $a \in A$ and $p \in \mathbb{Z}$ such that $R_a(c, p) = \text{true}$, if $f_a(c, p) = c'$ and $inv(c') = \text{true}$ then $\Delta\big(c, (a, p)\big) = c'$. The transition function is not defined in any other values of a, c and p.*

Note that the LTS of an action system leaves out those configurations for which the invariant does not hold.

3.1 Enabledness Abstractions

Now that we have defined the state space defined by an action system by means of its LTS, we need to define a proper level of abstraction in order to obtain a finite representation out of it. Experience so far indicates that grouping LTS states for which the same set of actions are enabled is an abstraction level that provides good compromise between size and precision. The definitions in this section are an adapted version of previous work by the authors [4].

Definition 3 (Enabledness Equivalence). *Given an action system $AS_{\mathbb{C}} = \langle A,\ F,\ R,\ inv,\ init \rangle$ and two configurations $c_1, c_2 \in \mathbb{C}$ we say that c_1 and c_2 are* enabledness equivalent *configurations (noted $c_1 \equiv_e c_2$) iff for every $a \in A$ $\exists\ p \in \mathbb{Z}\ .\ R_a(c_1, p) = \text{true} \Leftrightarrow \exists\ p' \in \mathbb{Z}\ .\ R_a(c_2, p') = \text{true}$.*

Notice that this definition is comparable to requiring simulation equivalence for just one step.

Given an LTS describing the semantics of an action system, we now define its enabledness-preserving abstraction as a finite non-deterministic state machine which groups the action system configurations according to the actions that they enable. Furthermore, this abstraction is able to *simulate any path* in the LTS describing the action system semantics.

Definition 4 (Enabledness-preserving Abstraction). *Given an action system $AS_{\mathbb{C}} = \langle A,\ F,\ R,\ inv,\ init \rangle$ and its LTS $L = \langle \mathbb{A},\ \mathbb{S},\ \mathbb{S}_0,\ \Delta \rangle$, we say that $M = \langle \Sigma,\ S,\ S_0,\ \delta \rangle$ is an* enabledness-preserving abstraction *(EPA) of $AS_{\mathbb{C}}$ iff there exists a total function $\alpha : \mathbb{S} \to S$ such that $\alpha(\mathbb{S}_0) \subseteq S_0$*

[1] For the sake of simplicity, without losing generality we set the number of parameters to only one and of integer type.

and for every $c \in \mathbb{S}$, action label a and parameter p such that $R_a(c,p)$ holds, then $\alpha(\Delta(c,(a,p))) \subseteq \delta(\alpha(c),a)$. Furthermore, given a pair of configurations c_1, c_2 on $\mathbb{S}$, then $c_1 \equiv_e c_2 \Leftrightarrow \alpha(c_1) = \alpha(c_2)$.

Where α is extended so that it can be also used as a function in $\wp(\mathbb{S}) \to \wp(S)$ in the natural way.

In order to construct an enabledness-preserving abstraction we first define the notion of *action set invariant*. Given a subset of actions $as \subseteq A$ of an action system $\mathrm{AS}_\mathbb{C}$, we wish to characterise all configurations c that satisfy the invariant inv and in which every action a in as is possible from c (there exists a parameter p such that the requires clause R_a of action a holds) and, importantly, in which every action a *not* in as it is *not* possible from c.

Definition 5 (Invariant of an Action Set). *Given an action system $AS_\mathbb{C} = \langle A,\ F,\ R,\ inv,\ init\rangle$, the invariant of a set of actions $as \subseteq \wp(A)$ is the function $inv_{as} : \mathbb{C} \to \{\mathrm{true}, \mathrm{false}\}$ defined as:*

$$inv_{as}(c) \overset{def}{\Leftrightarrow} inv(c) \ \wedge \bigwedge_{a \in as} \exists p.\ R_a(c,p) \ \wedge \bigwedge_{a \notin as} \nexists p.\ R_a(c,p)$$

We can now construct an enabledness-preserving abstraction of an action system by fixing the states to be the enumeration of all the possible action sets. We connect two action sets as and bs with a label a when there is a configuration c satisfying the invariant of as, such that when executing the action a, c evolves into a configuration that satisfies the invariant of bs.

Theorem 1 (EPA characterisation). *Given an action system $AS_\mathbb{C} = \langle A,\ F,\ R,\ inv,\ init\rangle$, then $M = \langle \Sigma,\ S,\ S_0,\ \delta\rangle$ is an EPA of $AS_\mathbb{C}$ where:*

1. $\Sigma = A$;
2. $S = \wp(A)$;
3. $S_0 = \{as \in S \mid \exists c \in \mathbb{C}.\ inv_{as}(c) \wedge init(c)\}$; *and*
4. *For all $as \in S$ and $a \in \Sigma$, if $a \notin as$ then $\delta(as,a) = \varnothing$, otherwise $\delta(as,a) = \left\{ bs \;\middle|\; \begin{array}{c} \exists c.\ inv_{as}(c) \ \wedge \exists p.\ R_a(c,p) \\ \wedge\ inv_{bs}\big(f_a(c,p)\big) \end{array} \right\}$.*

The proof for the theorem is by showing that, given the LTS L of an action system $\mathrm{AS}_\mathbb{C}$,

$$\alpha(c) \overset{\mathrm{def}}{=} \{a \in A \mid \exists\ p \in \mathbb{Z}\ .\ R_a(c,p) = \mathrm{true}\}$$

is a witness abstraction function such that every pair of configurations c_1, c_2 satisfies $c_1 \equiv_e c_2 \Leftrightarrow \alpha(c_1) = \alpha(c_2)$.

4. CONSTRUCTING EPAs

In this section we present the formal underpinnings behind the construction of the enabledness-preserving abstraction of an action system. We first present a construction algorithm which indicates which satisfiability queries need to be solved, but not how. We then discuss the impact of using a software model checker as a mean to solve the satisfiability queries prescribed by the algorithm.

4.1 Construction Algorithm

The algorithm presented in this section performs a BFS exploration of the enabledness state space, mitigating the need to exhaustively enumerate all the possible 2^n abstract states for a program with n actions (as item 2 of Theorem 1 otherwise suggests).

Definition 6 (Construction Algorithm). *Given an action system $AS_\mathbb{C} = \langle A,\ F,\ R,\ inv,\ init\rangle$ we build an EPA as $M = \langle \Sigma,\ S,\ S_0,\ \delta\rangle$ returned by the following procedure:*

```
1: procedure ConstructEPA(AS_C): M
2:     Σ ← A.
3:     S ← ∅
4:     δ(as, a) ← ∅   ∀as, a
5:     A⁻ ← {a ∈ A | ∀c. init(c) ⇒ ¬∃p. R_a(c, p)}
6:     A⁺ ← {a ∈ A | ∀c. init(c) ⇒ ∃p. R_a(c, p)}
7:     S_0^C ← {as ⊆ A | A⁺ ⊆ as, A⁻ ∩ as = ∅}
8:     S_0 ← {as ∈ S_0^C | ∃c. inv_as(c) ∧ init(c)}
9:     W ← queue starting with elements in S_0
10:    while there is a certain as as the head of W do
11:        W ← W − [as]
12:        S ← S ∪ {as}
13:        for each action a ∈ as do
14:            B⁻ ← {b ∈ A | ∀c, p. inv_as(c) ∧ R_a(c, p) ⇒ ¬∃p'. R_b(f_a(c, p), p')}
15:            B⁺ ← {b ∈ A | ∀c, p. inv_as(c) ∧ R_a(c, p) ⇒ ∃p'. R_b(f_a(c, p), p')}
16:            S^C ← {bs ⊆ A | B⁺ ⊆ bs, B⁻ ∩ bs = ∅}
17:            for each state bs ∈ S^C do
18:                if ∃c. inv_as(c) ∧ ∃p. R_a(c, p) ∧ inv_bs(f_a(c, p)) then
19:                    δ(as, a) ← δ(as, a) ∪ {bs}
20:                    if bs ∉ S and bs ∉ W then
21:                        W ← W ∪ [bs]
```

The transition function is initialised as empty for any input. The set A^- stores the actions that can never be enabled in any initial state. Conversely, A^+ holds those actions that have to necessarily be enabled in every initial state. A set of candidate initial states S_0^C is constructed by enumerating all the action sets that: *i)* exclude all the actions in A^-; *ii)* contain all the actions in A^+. All of the action sets in S_0^C are then tested in order to store in S_0 only those that comply with item 3 of Theorem 1. Notice that the more actions classified as necessarily enabled (or disabled) the smaller is the set of candidate initial states. Furthermore, this optimization takes a linear amount of operations in terms of predicates that need to be analysed.

Having determined S_0, the algorithm initialises a queue W of states (action sets) pending to be visited. Each time a given state as is visited, all of its enabled actions $a \in as$ are considered. The set B^- holds all those actions that can not be executed with any parameter after the execution of a from state as. Conversely, B^+ is the set of actions which have at least one parameter to be executed with after the execution of a from state as. The set of candidate destination states S^C is constructed in a similar way than S_0^C. All the states in this candidate set are considered in order to check each one of them and see if they can be actually reached by evolving as using a. Each time a new state is found, it is added to the pending states queue W.

This algorithm is, in the worst case, exponential in space with respect to the number of actions. However, the more actions we can classify as necessarily enabled (or disabled) in a particular state, the less candidate states the algorithm needs to consider. This optimisation makes running times come down significantly and allowed us to cope with real-life programs while keeping time down to a few minutes in the worst case, as we will argue in Section 5. Furthermore, the exploration nature of this algorithm makes it simple to parallelise using worker threads that share the pool of states to be visited.

We can now postulate that the outcome of this algorithm

is in fact an EPA compliant with Definition 4.

THEOREM 2. *Given an action system $AS_{\mathbb{C}}$, then M as built by Def. 6 is an EPA of $AS_{\mathbb{C}}$.*

The proof for this theorem is based on the fact that the abstraction constructed using Definition 6 is the reachable fragment of the abstraction presented in Theorem 1.

The algorithm in Definition 6 is a template that stipulates *which* validity checks need to be performed, but not *how* to solve them. Since validity checking is undecidable in general, we need to analyse the impact that uncertain answers in the validity checks may have on the algorithm's result.

For instance, when deciding if a certain action a needs to be included in the set A^-, the validity check $\forall c.\ \text{init}(c) \Rightarrow \neg\exists p.\ R_a(c,p)$ may return an uncertain answer. In this case it is safe to exclude the action a from the set A^- since there is no guarantee that it will necessarily be disabled on any initial state.

This has no impact on the algorithm's output, since A^- is only used to reduce the set of *potential* initial states. In other words, if an action a which is always disabled on any initial state is excluded from A^- due to an uncertain answer in the validity check, then it only makes the algorithm run slower; it does not affect the result. Similarly, the computation of sets A^+, B^- and B^+ is not affected by uncertainties. In fact, these sets could be set to $\varnothing$ without affecting the result.

On the other hand, the validity checks in lines 8 and 18 are critical for the result of the algorithm. Line 8 directly affects the set of initial states; line 18 affects the presence of transitions among states, therefore affecting which states of the EPA are reachable and deserve being explored. Uncertain answers in the validity checks in these two lines *do affect* the quality of the result, as indicated in the following theorem.

THEOREM 3. *Let $AS_{\mathbb{C}}$ be an action system, and M built by Def. 6 dealing with uncertainty as follows:* a) *If uncertain in line 8 then as is added to S_0.* b) *If uncertain in line 18 then the* ***then****-branch is executed.*

Then M satisfies a relaxation of the items in Theorem 1: i) *S_0 is a superset of the one in item 3; and* ii) *$\delta(as,a)$ is a superset of the one in item 4.*

A corollary for this result is that, in this context of uncertainty from the validity checks, the constructed M is a simulation of the EPA. In the next section, we present an operationalisation of the construction algorithm that deals with the fact that the elements of an action system are denoted by code fragments, and we will therefore present a novel approach to solve these validity checks by means of code reachability queries.

4.2 Construction via Code Reachability

The algorithm of Definition 6 performs logical manipulation of the mathematical functions defined in the program under analysis given by $\text{AS}_{\mathbb{C}}$ and therefore requires a decision engine. Since we want to be able to obtain EPAs from source code, in principle we do not have a symbolic representation of the mathematical functions it denotes (e.g., postconditions) and therefore, unlike previous work [4], we can not use a theorem prover (such as an SMT solver) in this context. In this section we explain how we can fulfil the tasks prescribed by each step of the construction algorithm resorting to code reachability queries.

For instance, given an action a, the step of line 5 requires an effective way of deciding the validity of $\forall c.\ \text{init}(c) \Rightarrow \neg\exists p.\ R_a(c,p)$. Consider the following procedure:

```
procedure QUERY-FOR-LINE-5(c : ℂ, p : ℤ)
    if CodeOf [init] (c) = false then
        return
    if CodeOf [R_a] (c, p) = true then
        TARGET
```

The TARGET statement is reachable by an execution of this procedure if and only if: *i)* there exits a starting configuration c which makes the initial predicate true; and *ii)* there exists a parameter p that makes the requires clause of a hold for the same configuration c. Formally:

$$\begin{aligned} \text{TARGET is reachable} &\equiv \exists c.\ (\text{init}(c) \ \wedge\ \exists p.\ R_a(c,p)) \\ \text{TARGET is unreachable} &\equiv \forall c.\ \neg\,(\text{init}(c) \ \wedge\ \exists p.\ R_a(c,p)) \\ &\equiv \forall c.\ \text{init}(c) \Rightarrow \neg\exists p.\ R_a(c,p) \end{aligned}$$

Meaning that the unreachability of the TARGET statement in the given procedure is equivalent to the validity of the predicate in line 5 of the algorithm. Following the discussion in the previous section, if the reachability decision engine is unable to provide a definite answer, it is interpreted as TARGET may be reachable, and then the action a is conservatively not added to the A^- set.

The encoding of the predicate in line 8 as a code reachability query is similar to the previous program.

The rest of the algorithm requires to decide the validity of similar predicates, however not any predicate can be solved using a code reachability query, since reachability can only encode safety properties. For instance, for the step of line 6, we need to decide the validity of $\forall c.\ \text{init}(c) \Rightarrow \exists p.\ R_a(c,p)$. This can not be encoded as a safety property since evidence of its validity must be provided in the form of a function that returns which p makes the requires clause hold for each configuration c.

The strategy we followed to overcome this problem is obtaining a pair of approximations of the original requires clause of action a: $\widehat{R_a}$ and $\widecheck{R_a}$. Formally, for every configuration $c \in \mathcal{C}$ and every parameter $p \in \mathbb{Z}$, then $\widehat{R_a}(c,p) \Rightarrow R_a(c,p)$ and $R_a(c,p) \Rightarrow \widecheck{R_a}(c,p)$.

Furthermore, each approximation must be rewritten as the conjunction of two predicates: one ranging over the configuration and another over the parameter. Formally:

$$\begin{aligned} \widehat{R_a}(c,p) &= \widehat{\text{SR}_a}(c) \ \wedge \text{PR}_a(p) \\ \widecheck{R_a}(c,p) &= \widecheck{\text{SR}_a}(c) \ \wedge \text{PR}_a(p) \end{aligned}$$

As we will show in the following section, it is frequent that the code of R_a evaluates a condition for the parameter and, independently, a condition over the configuration. Such cases are easy to handle. Typical cases where condition involves both parameter and configuration are membership or comparision queries. Usually, those could be exactly approximated by checking non-emptyness or non-nullity of substructures of configuration. Moreover, if non-trivial candidate approximations are somehow provided they can be verified correct. Checking the overapproximation is easy: it boils down into showing the imposibility of finding a configuration and parameter that satisfies the original clause but does not satisfy the overapproximation. The case of the underapproximation is a little bit tricker since it also requires a Skolem function code on the configuration for computing

candidate parameter. Given that function, checking the underapproximation boilds down into verifying that function always find a parameter p such that the configuration and p satisfies the original requires clause. Those verifications can be perfomed by means of reachability queries similarly to what is explained bellow. In the case study all this was pretty straightforward and it is not shown in detail due to space constraints.

Under this scenario, the validity of the sentence in line 6 is implied by $\forall c.\ \text{init}(c) \Rightarrow \exists p.\ \widehat{R_a}(c,p)$. Since $\widehat{R_a}$ can be split in two parts, this sentence can be rewritten as $\exists p.\ \text{PR}_a(p) \ \wedge\ \forall c.\ \text{init}(c) \Rightarrow \widehat{\text{SR}_a}(c)$. The validity of this conjunction can be solved using code reachability by means of two separate queries. The first one deals with the first part of the conjunction, and is solved by asking if the ERROR statement is reachable in the following procedure:

```
procedure QUERY-a-FEASIBLE(p : ℤ)
    if CodeOf [PR_a] (p) = true then
        ERROR
```

In fact, notice that this query does not depend on the value for the configuration c. If the ERROR statement was not reachable, then the a action could never be executed for any parameter (regardless of the configuration). In the rest of the paper we will assume that, given an action a, there is always at least one parameter that makes PR_a be true.

The second part of the conjunction, namely $\forall c.\ \text{init}(c) \Rightarrow \widehat{\text{SR}_a}(c)$, is solved by checking that the ERROR statement is unreachable in:

```
procedure QUERY-FOR-LINE-6(c : ℂ)
    if CodeOf [init] (c) = false then
        return
    if CodeOf [SR̂_a] (c) = false then
        TARGET
```

Notice that the *unreachability* of the TARGET statement is a sufficient condition to establish that a is enabled on every initial state. Therefore, we add a to the set A^+ only if we have conclusive evidence of unreachability. In other cases (i.e., reachability of TARGET or uncertain), we conservatively keep A^+ unchanged.

To decide the validity of the predicates in the rest of the algorithm, given an action set $as \subseteq A$ and a configuration c, we need to be able to determine whether $\text{inv}_{as}(c)$ holds. As requires clauses can be weakened and strengthened, we can calculate a weaker version without needing to refer to parameters:

$$\widetilde{\text{inv}_{as}}(c) \overset{\text{def}}{\Leftrightarrow} \begin{array}{l} \text{inv}(c) \ \wedge\ \bigwedge_{a \in as} \exists p.\ \text{PR}_a(p) \ \wedge\ \widetilde{\text{SR}_a}(c) \\ \wedge\ \bigwedge_{a \notin as} \neg\big(\exists p.\ \text{PR}_a(p) \ \wedge\ \widehat{\text{SR}_a}(c)\big) \end{array}$$

This can be simplified, since $\exists p.\ \text{PR}_a(p)$ is assumed to be true. Therefore, we can calculate this approximated state invariant using the following procedure:

```
procedure OVER-INVARIANT-OF-as(c : ℂ)
    ret ← inv(c)
    for a ∈ as do
        if CodeOf [SR̃_a] (c) = false then
            ret ← false
    for a ∉ as do
        if CodeOf [SR̂_a] (c) = true then
            ret ← false
    return ret
```

Using this procedure, the validity checks in lines 14 and 15 can be calculated using similar approximation strategies as the one used for lines 5 and 6. Notice that Theorem 3 allows us to make this conservative decisions without affecting the result of the algorithm.

We will focus on the validity check in line 18:

$$\exists c.\ \text{inv}_{as}(c) \ \wedge\ \exists p.\ R_a(c,p) \ \wedge \text{inv}_{bs}\big(f_a(c,p)\big)$$

In order to comply with Theorem 3, if in doubt, the sentence needs to be accepted as true, so that the **then**-branch of the **if** is executed. Therefore, and in order to translate the validity problem into a reachability query, we will check the validity of a weaker formula, using the approximations of the requires clauses. Concretely, we will try to decide the validity of the following sentence:

$$\exists c.\ \widetilde{\text{inv}_{as}}(c) \ \wedge\ \exists p.\ \text{PR}_a(p) \ \wedge\ \widetilde{\text{SR}_a}(c) \ \wedge \widetilde{\text{inv}_{bs}}\big(f_a(c,p)\big)$$

Since $a \in as$, then $\widetilde{\text{inv}_{as}}(c)$ includes $\widetilde{\text{SR}_a}(c)$ and this sentence is equivalent to:

$$\exists c.\ \widetilde{\text{inv}_{as}}(c) \ \wedge\ \exists p.\ \text{PR}_a(p) \ \wedge \widetilde{\text{inv}_{bs}}\big(f_a(c,p)\big)$$

The validity for this sentence can be derived by the reachability checking on the following code:

```
procedure QUERY-FOR-LINE-18(c : ℂ, p ∈ ℤ)
    if OVER-INVARIANT-OF-as(c) = true then
        if CodeOf [PR_a] (p) = true then
            c' ← CodeOf [f_a] (c, p)
            if OVER-INVARIANT-OF-bs(c') = true then
                TARGET
```

In case of unreachability of TARGET the transition is not added to the result. On the contrary, if it is reachable or if the decision engine is uncertain, then we assume that the TARGET statement is potentially reachable, and therefore the transition is still added to the result, complying with Theorem 3.

5. EXPERIMENTAL EVALUATION

In this section we comment on some of the aspects involved in the validation of our approach. In particular, we aim to answer the following research question:

> **R.Q.:** Is the proposed level of abstraction useful for validating code artefacts and identifying findings that relate to bugs in code and problems in expected or documented requirements?

In order to answer our research question we first implemented the algorithm of Definition 6 in the CONTRACTOR tool. We use the BLAST [2] software model checker to solve the code reachability queries of the previous section.

In the rest of this section we present the experiment design and the set of subjects used, together with the motivation for their selection. We then comment on the results related to answering the research question. This section ends with quantitative and qualitative analyses of the presented results, as well as a description of the threats to their validity.

5.1 Experimental Setting

In order to answer the previous research question, a series of case studies were conducted using the following design. First, a program under analysis is abstracted using the CONTRACTOR tool, generating an enabledness-preserving abstraction. Separately, behaviour requirements are procured. They may be manually generated by a third-party or derived from existing documentation.

Then, an expert reviewer compares the enabledness-preserving model with the behaviour requirements, yielding a list of suspicious differences between them. We will refer to these as *findings*. Finally, each finding is tracked back to the original program in order to confirm it is non spurious.

5.2 Subjects

The programs on which the studies were performed are the `PipedOutputStream`, `Signature` and `ListItr` from the Java Development Kit (JDK) 1.4 implementation; the `SMTP-Protocol` class from the RISTRETTO protocol-level Java mail client; and the `PCCRR` class was taken from a C# SpecExplorer protocol model.

Subject classes were included according to the following criteria: *i)* Classes that feature rich restrictions in the order in which the methods must be called. *ii)* Classes for which either behaviour documentation or manually-generated behaviour models can be found. *iii)* Classes that have already been analysed using techniques comparable to ours (e.g., [1, 11]). And *iv)* classes that are of industrial relevance.

The nature of the BLAST tool, which deals with C code, forces us to analyse programs that are written in that programming language. Therefore, all of these programs were manually translated to C in order for our tool to be able to analyse them. The requires clauses were set using the existing run-time checks in the source code.

For the first three classes, requires clauses did not require to be approximated, since they did not contain any atomic predicate which mentioned both a field and a parameter. The exception was the PCCRR class. In this program, a few actions had requires clauses which forced the value of a parameter to be exactly the same to the value of a class field. Since there is always an assignment to the parameter which is equal to the value of the field, setting $\widehat{R}$ and $\breve{R}$ to true could be used as an exact approximation of the original requires clause from an enabledness point of view.

Regarding the behaviour requirements, which were compared to the enabledness-preserving abstractions, they were obtained as follows. The `PipedOutputStream` EPA was compared against the official Java documentation (Javadoc). The `Signature` EPA was compared against the class Javadoc and against a manually-generated model made available by Dallmeier et al. [3]. The `ListItr` EPA was compared against a manually-generated model, which was constructed by a senior Java developer. The `PCCRR` EPA was compared against the reviewer's understanding of the protocol since the C# SpecExplorer model was undocumented. The `SMTPProtocol` EPA was compared against a manually-generated model made available by Dallmeier et al. [3].

5.3 Findings

In this section we report on the most relevant findings discovered while performing the case studies. All the generated models can be obtained from the CONTRACTOR tool Web site: `http://lafhis.dc.uba.ar/contractor`. All the components of the CONTRACTOR tool are freely available for download.

5.3.1 Java PipedOutputStream

The `PipedOutputStream` class instances is an implementation of an output stream that can be connected to a piped input stream to create a communications pipe. The piped output stream is the sending end of the pipe. More precisely, an instance of the `PipedOutputStream` class can engage in 4 different actions: `connect`, `write`, `flush` and `close`.

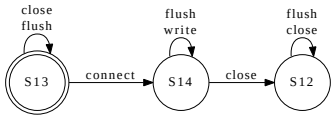

Figure 3: EPA of JDK 1.4 `PipedOutputStream`

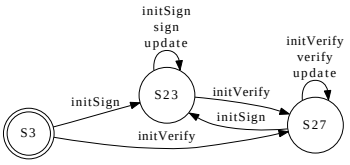

Figure 4: EPA of JDK 1.4 `Signature`

We produced a straightforward translation of the JDK 1.4 implementation of the `PipedOutputStream` to C. We also wrote the require clauses for the class methods as the necessary conditions to avoid exceptions being thrown when executing methods of the class. We then ran Contractor and obtained the following abstraction:.

The model in Figure 3 is the EPA for the `PipedOutputStream` obtained by CONTRACTOR. This abstraction shows to be an accurate representation of the Java official documentation. For instance, the Javadoc for the `connect` method says that "if this object is already connected to some other piped input stream, an `IOException` is thrown.". This is reflected in the EPA as the `connect` action is unavailable once a connection is established.

The documentation for the `close` method reads that after closure the "stream may no longer be used for writing". This is reflected in the transition from `S14` to `S12`, since the latter does not allow to perform the `write` operation.

More interestingly, the abstraction of Figure 3 shows a `close` loop transition on the initial state, which contradicts the Java documentation since it allows the following trace: `close` $\rightsquigarrow$ `connect` $\rightsquigarrow$ `write`, which exhibits the use of the writing operation after the pipe was closed. The reviewer analysed if this trace was legal in two ways: *i)* he first wrote an example program exercising this trace to see if it threw an exception; and *ii)* he then analysed the JDK implementation to see if there was any additional condition which might make the closure of a non-connected buffer throw an exception. The reviewer found that, despite the documentation says otherwise, the closure of unconnected piped output streams is legal and it produces no changes in the class fields whatsoever.

5.3.2 Java Signature

The Java `Signature` class is used to provide applications the functionality of a digital signature algorithm. There are three phases to the use of a `Signature` object for either signing data or verifying a signature: *i)* Initialization, with either a public key, which initializes for verification (`initVerify`), or a private key, which initializes for signing (`initSign`); *ii)* Updating, which updates the bytes to be signed or verified (`update`); and *iii)* Signing or verifying a signature on all updated bytes (`sign` and `verify` respectively).

The model in Figure 4 is the EPA for `Signature` obtained with CONTRACTOR on a C version of the JDK 1.4 implementation of class `Signature`, introducing split preconditions that prevented exceptions from being thrown and obtained the FSM in , which represents the resulting EPA.

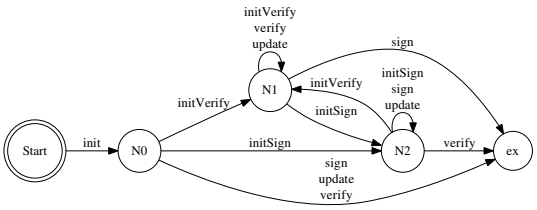

Figure 5: EPA of JDK 1.4 `manually generated model of Signature`

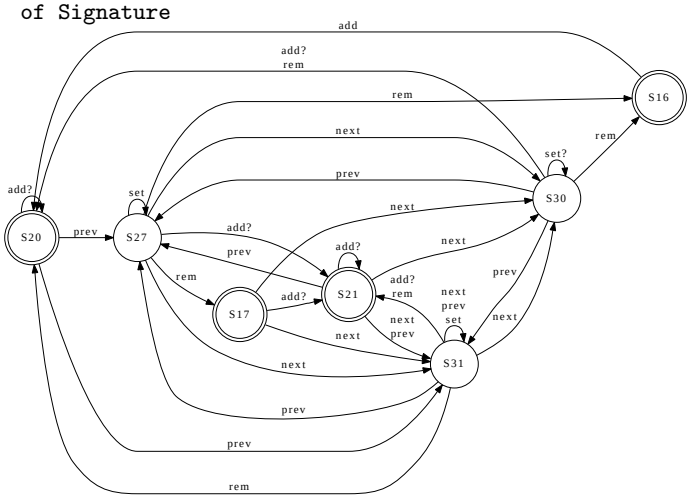

Figure 6: EPA of JDK 1.4 `ListItr`

The EPA obtained with CONTRACTOR was exactly the same as the manual model presented in [3]. This model clearly represents how an instance of `Signature` can only be in 3 different states: uninitialized (`S3`), initialized for signing (`S23`) or initialized for signature verification (`S27`). After checking the source code, the reviewer found that the implementation stores this information in an integer variable named `state`, which takes values from the set {`UNINITIALIZED`, `SIGN`, `VERIFY`}.

Our abstraction also proved to be a faithfull representation of both the manually-generated model presented in [3] (see Figure 5), as well as of the restrictions imposed by the official Java documentation.

5.3.3 Java List Iterator

The Java List Iterator (`ListItr`) provides functionality to go through the elements stored in a List.

It is initialized passing both the target list and the initial index from which the iteration begins. If the end of the list has not been hit, the iterator can retrieve the `next` element. Conversely, it can retrieve the `prev` element if the current index is not 0.

The `add` operation can be invoked at any moment, and it incorporates a new element at the current position. The `rem` and `set` methods operate on the last element that has been retrieved: the first one removes it from the list; the other replaces it by another element.

The abstraction of Figure 6 is the EPA obtained by CONTRACTOR when analysing the JDK 1.4 Java List Iterator implementation over an `ArrayList` where the split preconditions were set so that no exceptions were thrown. Every state in it represents an interesting situation to which an iterator can evolve. There are 4 initial states:

`S16` : the `add` operation is the only available action. We are iterating over an empty list.

`S17` : the `prev` operation is disabled, so the cursor is at the beginning of the list. The `set` and `rem` operations are also disabled, so this means that an element hast not been just retrieved. The `next` operation is enabled, so the list is not empty.

`S20` : just like the previous state, but with the cursor pointing at the end, so `prev` is enabled but `next` is not.

`S21` : the `set` and `rem` operations are disabled, so there has not been a retrieved element. The cursor is pointing at a position in the middle of a list, so both `prev` and `next` are enabled.

The states `S27`, `S30` and `S31` are like states `S17`, `S20` and `S21` respectively. The only difference between them is that in the former states an element has just been returned, so the `rem` and `set` operations are also enabled. Notice that while producing this EPA BLAST was uncertain in a number of transitions, which are suffixed with a "?" symbol. CONTRACTOR reported that the cause of this uncertainty is that the invariant may not be preserved. A finer-grained manual analysis revealed that the invariant is not violated, however BLAST is not able to prove this.

A senior Java developer with more than 8 years of experience (including experience with formal models) manually generated a behaviour model, shown in Figure [**?**] by analysing the JDK implementation for the list iterator. During this creation process, the developer executed a number of usage scenarios to refine his understanding of the code.

When comparing this manually-generated model with the EPA, the reviewer discovered that the overall level of abstraction was comparable to that of enabledness: more than half of the states in the manually-generated model were present in the EPA. Furthermore, there were 2 states in the manually-generated model which were enabledness-equivalent. This is because the developer decided to separately consider the cases in which the iterated list had exactly one element. Finally, there were 3 states in the manually-generated model which were not traceable to states in the EPA. When further analysing the conditions implied by these states, the reviewer discovered that they were exhibiting spurious behaviour and were accidentaly introduced by the developer, due to his misunderstanding of the requirements.

5.3.4 PCCR Framework

The *Peer Content Caching and Retrieval* (PCCR)[2] system is a P2P-based distribution framework designed to reduce bandwidth consumption in wide area networks. The key feature is that it which allows clients to retrieve content from *distributed caches* when available, instead of *content servers* which are generally located remotely. In order to increase the local availability of content, clients also serve as caches.

This framework is defined by two protocols, one of which (PCCRR) is used for querying the server for the availability of certain content and retrieving it.

Based on the quality process, model-based testing approach described in [10], the reviewer analysed the program that defines the SpecExplorer model used to guide the testing process of the protocol's client side. In a few words,

[2] `http://msdn.microsoft.com/en-us/library/dd304175(PROT.13).aspx`

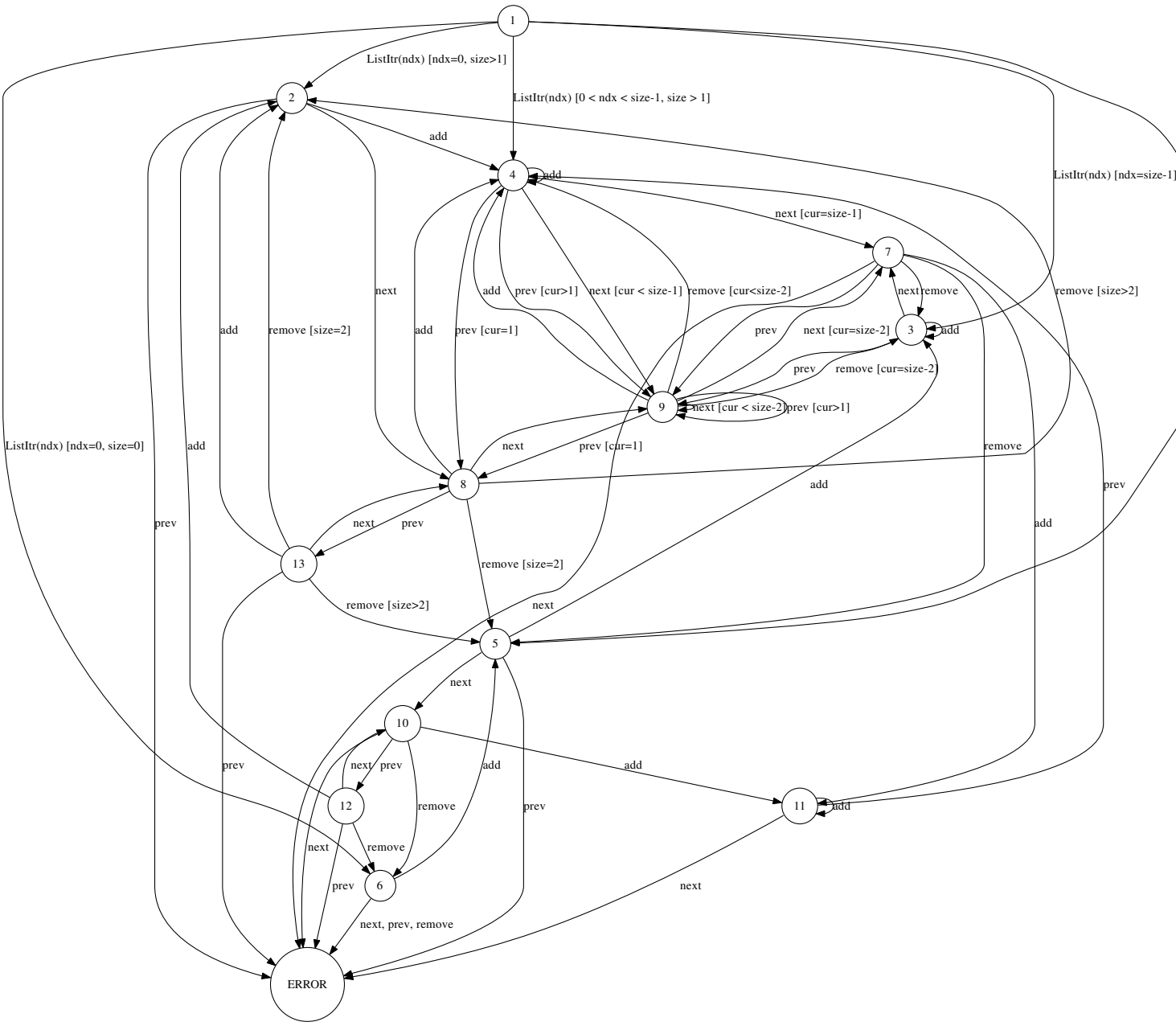

Figure 7: EPA of JDK 1.4 `ListItr`

a SpecExplorer model is a C# class consisting of methods that are interpreted as guarded rules defining a rich action machine. These rules are used to stimulate the system under testing and check its answers. In this case, the model program could be regarded as an abstract implementation of the server side.

Concretely, the PCCRR protocol model defines the following actions:

`GetSutPlatform` establishes which operating system runs on the client to be tested.

`InitSut` initializes the client to be tested.

`RcvNegoReq` indicates that the server has received a message with the protocol versions supported by the client.

`SndNegoResp` is the same as the previous one, but the message is sent by the server.

`RcvGetBlkList` indicates that the server has received a message requesting the hashes for set of blocks which the client is interested in.

`SndBlkList` makes the server send the hashes for the requested blocks it possesses to the client.

`SndBlkListAb` is the same as the previous one, but with a response indicating the request was not consistent.

`RcvGetBlk` indicates that the server has received a request for a particular block, indicated by its hash.

`SndBlk` is the action by which the server sends the requested block to the client.

Where the `Snd`-prefixed actions are those that correspond to messages controlled by the program, while the rest of the actions correspond to messages that the program monitors.

It is worth mentioning that there are no ordering restric-

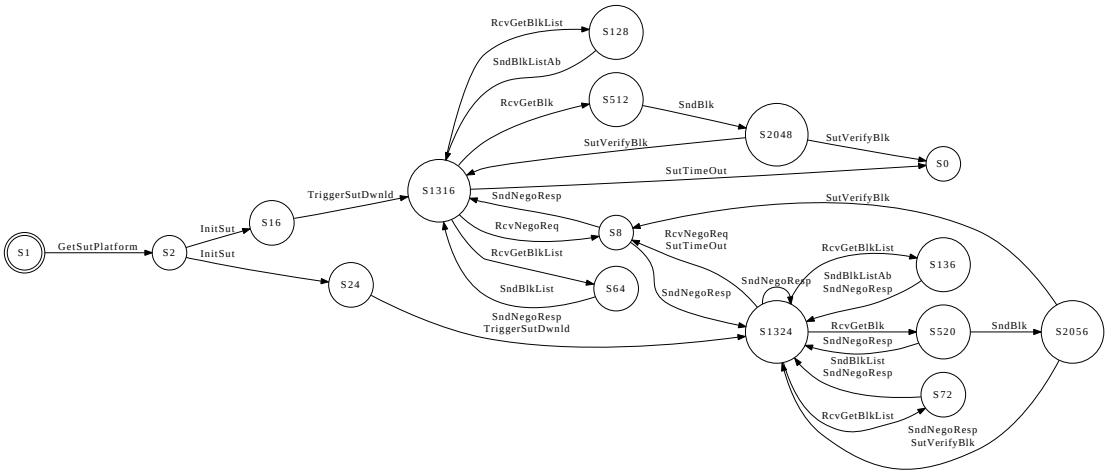

Figure 8: First FSPA of the MS-PCCRR server side

tions between the requests that come from the client once the establishment of the connection is done. For example, a client may first ask for the supported versions of the server, then for the list of packages and finally decide not to download anything. Another client may directly ask for a specific package without even asking for the package list.

The program also has the following *control* actions, which are communications with the client under test that are not defined in the protocol documentation. These are used to control the progress of the testing process itself:

`TriggerSutDwnld` makes the client under test request a download to the server.
`SutTimeOut` indicates that the client has timed out.
`SutVerifyBlk` verifies that the client has correctly received the requested block.

We translated the C# class to C, using the guards for the SpecExplorer rules as requires clauses for the CONTRACTOR input. In this case we needed to relax 3 preconditions by hand in order to comply with the requisite of split preconditions.

The first EPA obtained with CONTRACTOR, which had 16 states, which can be seen in Figure 8, was relatively big but still much smaller that the model with 844 states produced by SpecExplorer during an exploration of the PCCRR state space. The reviewer analysed this abstraction and found that starting from the initial state (`S1`). The `InitSut` initialisation action showed non-deterministic behaviour. This was suspicuous and led to the discovery that as it can evolve to `S16` and `S24`. We checked the code for this action and we discovered that it has a single boolean parameter called `isTestingNegotiation`, which is stored in a boolean field named `isTestingNego`. The reviewer then searched for other appearances of this the `isTestingNego` field and found that when it this true then a negotiation response the `SndNegoResp` action is enabled. In the EPA depicted in Figure 8, this issue is manifested as it featured a quasi-partition of the states into two sets which are only connected by 2 transitions. states `S8` and `S1316`.

A negotiation response action is always enabled in one of the model fragments and always disabled in the other. Furthermore, this is the only difference between the 2 sets of states.

This issue appears to be a case of a weak dispatch condition of the `SndNegoResp` action, which might result in the generation of test cases where the program behaves differently than the client under test is expecting.

In order to get a better understanding of the model code, we decided to fix the platform to be not Windows-based, getting the abstraction in Figure 9.

CONTRACTOR was then ran over a modified version of the original code, obtained by eliminating the `isTestingNego` field, getting an abstraction featuring 10 states. This second abstraction allowed the reviewer to find another unknown issue in the program: an operation `SutTimeOut` which should only be triggered when the client is idle has a weak requires clause which could lead to false positives if the action is executed with a package still on-the-fly. This second abstraction also reflected the fact that, once the connection is established, there are no ordering restrictions between the messages that the client may send.

5.3.5 SMTPProtocol

The `SMTPProtocol` class is a Java implementation of an SMTP protocol client. CONTRACTOR was ran and obtained its enabledness-preserving abstraction (see Figure 10). When compared to the manually-generated model in [3] ((see Figure 11)) the reviewer discovered that the EPA was more permissive. In particular, the manually-generated model reflected a number of method ordering restrictions, such as requiring mails to be initiated (`mail`) before recipients could be added (`rcpt`).

On the other hand, the EPA constructed allowed recipients to be added anytime, as long as the connection with the server was established. This lack of restrictions in the EPA is caused by the `SMTPProtocol` implementation, which only keeps track of a single variable which indicates if the client is connected or not. When the client is connected, the implementation acts as a pass-through of the user requests to the server and delegates the enforcement of invocation ordering restrictions to the server. This pass-through behaviour of the SMTP client implementation relies on a well implemented SMTP server to work properly.

On the other hand, the manually-generated model presents a flattened view which includes action ordering restrictions which are actually enforced by the (presumable

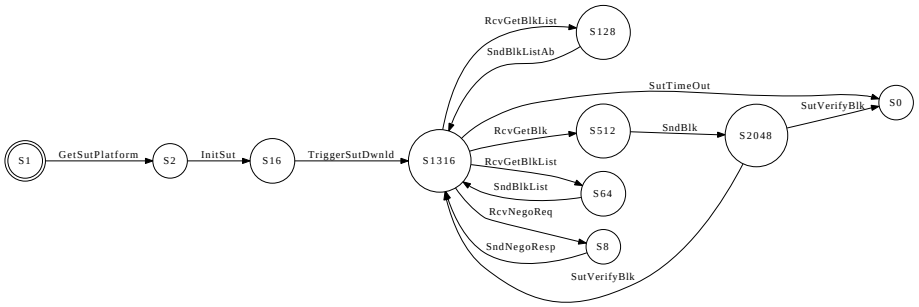

Figure 9: Second FSPA of the MS-PCCRR server side (eliminated the `isTestingNego` field)

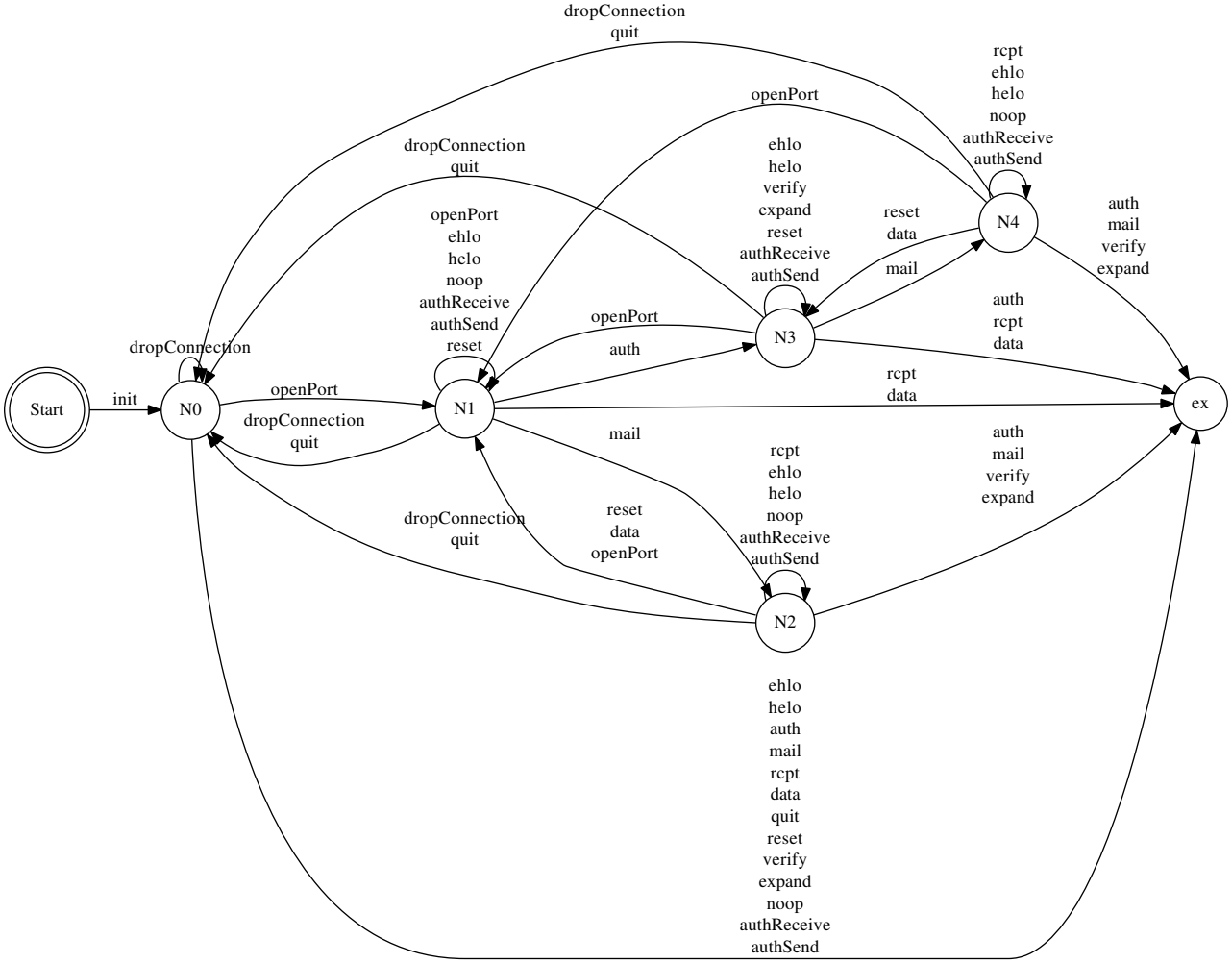

Figure 11: EPA for SMTP protocol client class

well implemented) SMTP server.

5.4 Quantitative Analysis

The case studies presented on this section were ran on an Intel i7 (hyper-threaded quad-core) computer with 4 GB of RAM. The algorithm was executed with 8 worker threads running in parallel performing the Blast queries.

Table 1 presents a quantitative view of the performed case studies. Engine certainty accounts for the percentage of successful (i.e., certain) answers from Blast.

As we can observe, running times do not only depend on the number of actions, but also on the size of the abstraction, as can be seen when comparing the two versions of `PCCRR`.

It is worth mentioning that the reachable fragments of the EPAs constructed with Contractor feature significantly less states than the complete $2^{|A|}$ enabledness-based state space. For instance, the `ListItr` EPA has 7 states out of 32; the second `PCCRR` EPA has 10 states out of 4096.

We also want to study if the optimisation presented in the construction algorithm provides some benefit. In particular,

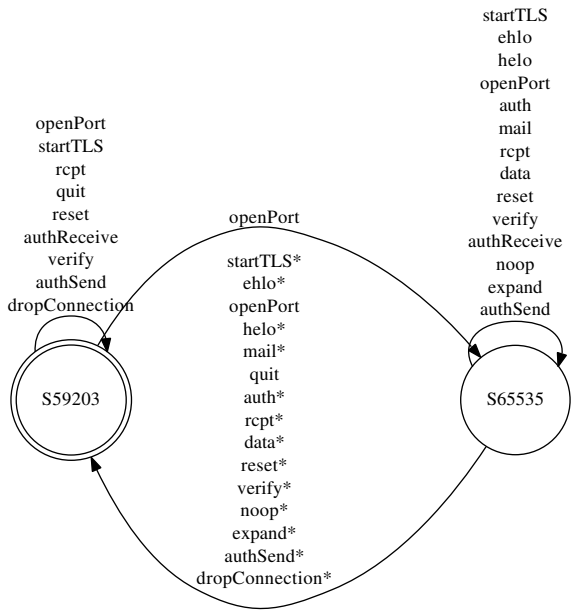

Figure 10: EPA for SMTP protocol client class

Input		Tool execution		Output	
Name	**Actions**	**Time (secs.)**	**Engine certainty**	**States**	**Transitions**
`List (buggy)`	3	6	100%	3	7
`List (fixed)`	3	7	100%	3	8
`PipedOutputStream`	4	8	100%	3	8
`Signature`	5	13	100%	3	10
`ListItr`	5	314	97.5%	7	32
`PCCRR (first)`	12	628	100%	16	34
`PCCRR (second)`	12	398	100%	10	14
`SMTPProtocol`	16	288	96.7%	2	39

Table 1: Case studies summary

we want to analyse is the overhead of computing A^+/A^- is compensated later when less actions has to be considered. Table 2 shows the CONTRACTOR tool running times when the A^+/A^- optimisation is disabled. In this case, in every abstract state, every enabled action and every possible reaching state is considered by the engine. For the smaller examples (up to 5 actions) disabling this optimisation is almost always beneficial; in bigger cases however, the unoptimised version runs up to 5 times slower. This behaviour clearly reflects the fact that computing the A^+/A^- sets is linear, and is therefore amortized.

5.5 Qualitative Analysis

In order to provide an answer to our research question, we will argue that the EPAs created by our technique convey a representation of the behaviour which is tractable by human inspection and meaningful with respect to elements in the program under analysis.

For instance, the `PipedOutputStream` case study shows the potential of our approach to contrast the descriptive official documentation of an artefact with its current prescriptive implementation, which is what ends up being executed. In this case the reviewer found interesting behaviour which is officially undocumented but still legal.

In the `Signature` case study, our approach proved useful to easily trace elements in the abstraction back to source code elements such as variable definitions or value ranges.

The `ListItr` case study allowed us to discover that the states in a manually-generated model can sometimes be easily traced to enabledness-based states. Furthermore, the automatic nature of our approach prevents the developer from making mistakes when creating this kind of model. This case study also showed that, even in the presence of (a small number of) uncertain answers from the reachability engine, our approach successfully builds a non-trivial abstraction that is still amenable for validating and understanding the program under analysis.

In the `SMTPProtocol` case study, our abstraction clearly reflected the fact that the client-side implementation relied on server-side action ordering restriction checks; something which was not explicit in a previous manually-generated model available in literature.

Finally, the `PCCRR` case study showed how the automated construction of an EPA was helpful in identifying previously unknown relevant problems in an industrial model program.

It is worth mentioning that, even when the abstractions include spurious transitions due to undecidability and requires clauses approximations, the findings discovered by the reviewers, which are the ones we report here, were non spurious.

To conclude, the level of abstraction of the resulting EPAs has showed to be useful for tracing back both transitions and states to the source code, providing a helpful aid to gaining insight on the behaviour of the code and performing bug finding related tasks.

Input	Tool execution	
Name	**Optimised (secs.)**	**Unoptimised (secs.)**
`List (buggy)`	6	4
`List (fixed)`	7	4
`PipedOutputStream`	8	6
`Signature`	13	9
`ListItr`	314	369
`PCCRR (first)`	628	1078
`PCCRR (second)`	398	698
`SMTPProtocol`	288	1690

Table 2: Comparing running times with and without A^+/A^- optimisation

5.6 Threats to Validity

As any study, the results presented in this section are subject to threats to validity. We distinguish between threats to internal, external and construct validity.

Threats to external validity concern our ability to generalise the results of our study. We cannot generalise the results since the scope of our study is small (a sample of 5 programs). For instance, we have translated and analysed 2 of the 3 complete models presented in [3].

Scalability of our tool to cope with larger classes still remains a question. However, the scalability of our methodology relies on the scalability of the underlying software model checker tool that we use. Furthermore, the complexity of our algorithm is not as affected by the number of lines of the API implementation under analysis, but by the *number* of actions of the API which modify the state. Finally, our approach can be easily adjusted in a precision vs. scalability trade-off by introducing time-outs when calling the software model checker.

Although interesting findings were revealed by the abstraction, there may be issues at the method body level which could not revealed by the chosen granularity of action labels. It is possible to use labels to denote pairs of

request/response-kind by providing requires clauses that ensure method is executed and the expected type of response is yielded. Such a denotation would yield a finer grained abstraction that could reveal more issues. We plan to explore this in future work.

We are also biased in the selection since we have deliberately chosen programs with a rich action ordering restrictions. Simpler programs would yield trivial models which would not be as useful.

Threats to internal validity concern our ability to draw conclusions between our independent and dependent variables. The C translations of the subject programs or the manually-generated models may be incorrect. We minimise this risk by mostly using material that has been previously used by other authors, as well as making publicly available all the new material.

Threats to construct validity concern the adequacy of our measures for capturing dependent variables. The reviewer may have made mistakes when comparing the enabledness-preserving abstractions with the behaviour requirements. We believe that making all the material publicly available mitigates this threat.

6. RELATED WORK

In [4] we studied the enabledness-based abstractions and their potential for contract specifications validation. In that work we leveraged the fact that the input contract was a first-order logic description of the artefact under analysis, therefore amenable to symbolic manipulation with SMT solvers. This previous technique could have been applied in the context of analysing an API implementation by inferring a specification. However, precise postcondition inference is known to be hard in practice [12]. Instead, in this paper we deal directly with source code artefacts, which are more complex than contract specifications since they are not declarative and they exhibit features such as loops, memory management and procedure invocations, among others. In order to cope with this complexity, as we presented in Section 4, we introduced theorem 3 together with the over and underapproximated requires clauses, which allowed us to use a software model checker for the EPA construction.

Our technique is related to approaches that synthesize typestates [16, 5] or interfaces [1, 7, 11] out of a program: any sequence of methods that is not accepted by our abstraction will not be allowed by a program. However, in typestate and interface synthesis approaches the aim is modular verification, rather than validation. This imposes a safety requirement which tends to make their abstractions overly restrictive in terms of the model behaviour. Permissivness is possible only at the cost of assuming certain conditions over the artefacts they analyse, for instance the algorithms in [7, 11] guarantee correctness only when the library's internal state is finite. Examples with unbounded internal state are treated by limiting the number of observed exceptions and changing the signature of methods, as can be seen in the interface of Figure 6 of [1]. This abstraction for the `ListItr` example aims at client safety for only 2 of the 5 operations, and considers only 1 out of 3 exception types. Obtaining a safe interface for the complete class, considering all the actions and exceptions would have produced a trivial abstraction which would be of little use for validation purposes.

Our work can be considered an instantiation of the predicate abstraction [17] framework. In this setting our work is related to techniques that construct abstract state graphs from infinite state systems (e.g., [13, 8, 9]). However, these techniques aim at verification or generation of test cases rather than validation, hence the level of abstraction, the size of the resulting model and the challenge of traceability with the original artefact vary. For instance, even setting the input predicates in [8] to model the enabledness conditions of actions, the output would be likely too large for manual inspection (see [4] for further discussion). Notably, the setting in [9] admits producing the same abstraction than ours for testing purposes but the approach is to underapproximate it by finitely bounding the artefact under analysis.

A level of abstraction somewhat related to that of enabledness has been used in [14]. The authors quotient the state space of a class based on the result of its parameterless boolean observers. The abstraction is not meant to represent behaviour (e.g., it does not define transitions between states) but to define goals for test coverage criteria. The abstraction is then highly dependent on the quantity and quality of observers which may not have a correspondence with requires clauses, therefore yielding a different result from ours.

Our approach relates to the mining of temporal specifications (e.g., [6, 15, 3]), which aims at producing, from traces, a finite state automaton that describes how a set of operations is used. Unlike our approach, these techniques aim at inferring an specification which is used for test case generation or verification. Furthermore, mining techniques have a dynamic flavour, and thus heavily depend on the quality of the traces used as input. The inferred models may have both under and overapproximations of the artefact under analysis behaviour. On the other hand, our technique *statically* yields a model that is an abstraction of the program's source code, considering all possible paths.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a novel technique aimed at constructing abstract behaviour models out of a program's source code. The model is built using an enabledness-preserving level of abstraction, which is well suited for validation and debugging of the original artefact. We implemented an algorithm to build such behaviour models which relies on the use of a software model checker as a decision procedure to solve code reachability queries. We showed how the obtained models can be used to gain insight into the intended behaviour of a program, to discover defects in it, and to fix them by tracing them back to the original code.

We plan to analyse our approach in more industrial artefacts to further asses its scalability and validity .

Two candidate areas for providing a better tool support are: *i)* enhanced means of model visualisation, and *ii)* debugging by means of explorations over ground values.

Besides, we conjecture it is easy to integrate different types of software analysis tools in the construction algorithm. For instance, assertion verification techniques like [12] could be used to solve the validity checks, instead of reachability decision tools.

We also plan to work in mitigating the annotation burden by including automated techniques for invariants and requires clauses mining.

Acknowledgments. The work reported herein was partially supported by CONICET, UBACyT X021, PIP112-

200801-00955KA4, and PICT-PAE 37279.

8. REFERENCES

[1] R. Alur, P. Černỳ, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL '05*, pages 98–109, 2005.

[2] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9:505–525, 2007.

[3] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA 2010*, 2010.

[4] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *TSE*, in press 2010.

[5] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *PLDI '01*, pages 59–69, 2001.

[6] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE '08*, pages 51–60, 2008.

[7] D. Giannakopoulou and C. Păsăreanu. Interface generation and compositional verification in JavaPathfinder. In *FASE '09*, pages 94–108, 2009.

[8] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV '97*, pages 72–83, 1997.

[9] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA '02*, pages 112–122, 2002.

[10] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *STVR*, (in press).

[11] T. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/FSE '05*, pages 31–40, 2005.

[12] G. Leavens, K. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[13] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *STOC '92*, pages 264–274, 1992.

[14] L. Liu, B. Meyer, and B. Schoeller. Using contracts and boolean queries to improve the quality of automatic test generation. In *TAP '07*, pages 114–130, 2007.

[15] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08*, pages 501–510, 2008.

[16] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 12(1):157–171, 1986.

[17] T. Uribe. *Abstraction-based Deductive-algorithmic Verification of Reactive Systems*. Stanford University, Dept. of Computer Science, 1999.