A

Enabledness-based Program Abstractions for Behaviour Validation

GUIDO DE CASO, Universidad de Buenos Aires
VICTOR BRABERMAN, Universidad de Buenos Aires
DIEGO GARBERVETSKY, Universidad de Buenos Aires
SEBASTIAN UCHITEL, Universidad de Buenos Aires and Imperial College

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 approach addresses this problem by generating abstract behaviour models which resemble typestates. These models are statically computed and encode all admissible 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/Program Verification**]: Validation

General Terms: Design, Algorithms, Documentation

Additional Key Words and Phrases: Source code validation, enabledness abstractions

ACM Reference Format:
de Caso, G., Braberman, V., Garbervetsky, D., and Uchitel, S. 2011. Program Abstractions for Behaviour Validation. ACM Trans. Embedd. Comput. Syst. V, N, Article A (January YYYY), 47 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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 [Beckman et al. 2011]. 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 [Strom and Yemini 1986] from API implementations which are then used to verify if client code conforms to the implemented protocol [Alur et al. 2005; Dallmeier et al. 2010]. Such approaches, however, address only part of the problem: they assume the code from which the type-

The work reported herein was partially supported by CONICET, UBACyT 20020100100813, UBACyt 20020090300064, PIP112-200801-00955KA4, PICT 2010-2351, PICT-PAE 37279 and FP7-PEOPLE-2011-IRSES MEALS.
Author's addresses: G. de Caso, V. Braberman, D. Garbervetsky and S. Uchitel, Departamento de Computación, FCEyN, Universidad de Buenos Aires; S. Uchitel is also with the Department of Computing, Imperial College, London, UK.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

state 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. We believe that even in the absence of a full-fledged formal specification, the developer still has an informal understanding of the class he is building and the desired requirements that it has to meet. We usually refer to this informal understanding as *mental model* [de Caso et al. 2010].

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 used for verification [Uribe 1999]. More concretely, abstractions aimed at automated verification are generally very detailed (in terms of number of states, transitions and extra annotations/data). Automated tools are good at dealing with these rich abstractions, but humans can get easily confused or overwhelmed. Manual inspection therefore requires smaller yet meaningful abstractions.

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 a way to characterize abstraction properties: a typestate is *safe* [Alur et al. 2005] if no call sequence violates the library's internal invariants; it is *permissive* if it contains every such sequence. Previous approaches have aimed (e.g., [Henzinger et al. 2005]) at modular program analysis using typestates 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 contributions of this work can be summarised as follows:

— An algorithm to automatically and statically construct enabledness-preserving behaviour models from programs equipped with requires clauses.
— The implementation of this algorithm into a publicly-available practical tool that analyses C programs.
— The evaluation of our implementation applied to a series of real classes on which issues were found.

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 and its implementation into a practical tool (Section 4). We then 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;
3  } node;
4  typedef struct list {
5    int size; node* first;
6  } list;
7
8  list* l;
9
10 int inv() {
11   return l==NULL || l->size >= 0;
12 }
13
14 int List() {
15   l = (list*) malloc(sizeof(list));
16   if (l == NULL) return 0;
17   l->size = 0; l->first = NULL;
18   return 1;
19 }
20
21 int add_req() { return l!=NULL; }
22 int add(int data) {
23   node *tmp = l->first;
24   while (tmp->next != l->first)
25     tmp = tmp->next;
26   tmp->next =
27     (node*) malloc(sizeof(node));
28   if(tmp->next == NULL) {
29     l = NULL; return 0;
30   }
31   tmp->next->data = data;
32   tmp->next->next = l->first;
33   l->size++; return 1;
34 }
35 int remove_req() {
36   return l!=NULL && l->size > 0;
37 }
38 void remove(){
39   node* new_first = l->first->next;
40   free(l->first);
41   l->first = new_first;
42 }
43
44 int destroy_req() {
45   return l!=NULL;
46 }
47 void destroy() {
48   node* current;
49   node* tmp;
50   current = l->first;
51   l->first = 0;
52   while(current != 0) {
53     tmp = current->next;
54     free(current);
55     current = tmp;
56   }
57   l = 0;
58 }
```

Fig. 1: A singly-linked list C implementation

Consider the C source code of Figure 1, which implements a singly-linked integer 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 a system invariant (`inv()`) and a requires clause for each of its operations (`add_req()`, `remove_req()`, and `destroy_req()`).

How can we validate if this implementation provides the intended functionality when there is no formal and validated model of the intended functionality to compare

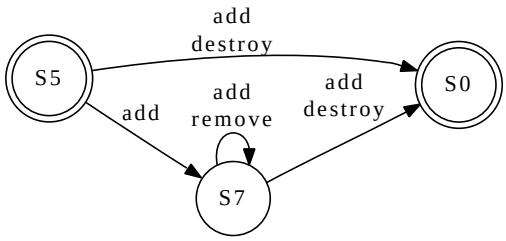

Fig. 2: Singly-linked list enabledness abstraction

against? As mentioned previously, one strategy would be to write a specification (or use an existing one) and verify the implementation against it using techniques such as testing, model checking or refinement checking. Such strategies can be effective at finding faults, however, they require a specification and shift the validation problem as it is now the specification itself that must be validated.

We propose to automatically extract a behaviour model such as the one shown in Figure 2. In this model we abstract the concrete state space of the singly-linked list based on the set of operations the concrete states enable, that is, the set of operations for which their requires clauses hold. Note that state labels are automatically generated, are only for reference, and have no meaning. The abstract state `S5` groups concrete states that allow execution of `add` and `destroy`. Abstract state `S7` groups concrete states that allow `add`, `remove`, and `destroy`. And `S0` groups all concrete states that do not allow any operation. Note that *initial states* are marked with a *double circle*.

The behaviour model in Figure 2 is permissive. Every legal operation sequence on the list implementation is included on the model's language. This permissiveness is succinctly obtained at the cost of sacrificing safety. There are operations sequences in the model's language which are not legal on the list implementation. Notice that in general it is not possible to have a finite state machine that safely and permissively captures the behaviour of an implementation, since only regular languages can be encoded using finitely many states. In this particular case, the sequence `add` $\rightsquigarrow$ `remove` $\rightsquigarrow$ `remove` is part of the model's language but it is not legal.

According to our previous experience [de Caso et al. 2010], sacrificing safety for the sake of obtaining a finite (and hopefully compact) behaviour model enables human inspection. Had we decided to construct a finite and safe behaviour model, we could have only allowed a single call to `remove`, since a finite model can not keep track of how many times this operation has been invoked with respect to `add` invocations.

More concretely, the 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 this 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 question arises, why choose this particular level of abstraction? While it may seem overly specific to present just a single level of abstraction, our claim is that the enabledness-based level of abstraction presents a good size/precision ratio in terms of facilitating developer-in-the-loop API validation. This level of abstraction not only yields a compact finite abstract model from an infinite concrete state space, but also allows tracing back concerns to the source code for identifying and fixing problems in the latter. That said, we do not discard the possibility that other abstraction levels could also prove useful in helping developers during validation tasks.

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

As we mentioned before, the object under analysis for our technique is the source code of a program. In order to abstract ourselves from the concrete details of the execution model for different programming languages, we will formally define API elements as transformations over system *configurations*. A configuration encompasses the API internal state and the elements in the heap. In the rest of this work, $\mathbb{C}$ will denote the set of all possible configurations.

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 system 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 3.1 (*Action System*). An *action system* is a structure of the form $AS = \langle Act,\ F,\ R,\ inv,\ init \rangle$, where:

— $Act = \{a_1, \ldots, a_n\}$ is a finite set of *action labels*.
— F is an Act-indexed set of *functions*. For each action label a, the function $F_a : \mathbb{C} \times \mathbb{Z} \to (\mathbb{C} \cup \perp)$ takes a configuration and an integer parameter and has two possible outcomes: *i)* either it transforms the configuration, or *ii)* it does not terminate (represented by the $\perp$ symbol).
For the sake of simplicity, without losing generality we will restrict ourselves to consider functions with a single integer parameter.
— R is an Act-indexed set of *requires clauses*. For each action label a, the requires clause $R_a : \mathbb{C} \times \mathbb{Z} \to \{\text{true}, \text{false}\}$ indicates if the action a is enabled for the given configuration and parameter.
— $inv : \mathbb{C} \to \{\text{true}, \text{false}\}$ is the action *system invariant*.
— $init : \mathbb{C} \to \{\text{true}, \text{false}\}$ is the *initial condition*, which indicates if a given configuration is an initial configuration for the action system.

In the rest of this paper, we assume that the provided invariant correctly and precisely characterises legal instances of the action system. In Section 4.5 we discuss the consequences of the violation of this assumption.

Example 3.2 (*List Action System*). A possible action system for the functional model of the list C implementation of the previous section is $AS = \langle Act,\ F,\ R,\ inv,\ init\rangle$ where:

— $Act = \{\texttt{add}, \texttt{remove}, \texttt{destroy}\}$. This set of actions indicates the names of the actions exposed in the *public interface* of the list implementation.
— $F = \{F_{\texttt{add}},\ F_{\texttt{remove}},\ F_{\texttt{destroy}}\}$. Where these functions are the semantic interpretation of the corresponding C functions in Figure 1.
— $R_{\texttt{add}}$ yields true only for configurations on which the `l` variable is not `NULL`.
— $R_{\texttt{remove}}$ yields true only for configurations that have a non-null `l` variable whose `size` field is positive.
— $R_{\texttt{destroy}}$ is the same as $R_{\texttt{add}}$.
— 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 to `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 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 CodeOf$[R_{\texttt{add}}]$ = `return head != NULL;`. Similarly, CodeOf$[F_{\texttt{add}}]$ is the fragment of lines 22–34 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 3.3 (*Action System Semantics*). Given an action system $AS = \langle Act,\ F,\ R,\ inv,\ init\rangle$, we say that its *semantics* is provided by an LTS $L = \langle \mathbb{A},\ \mathbb{S},\ \mathbb{S}_0,\ \Delta\rangle$ satisfying that $\mathbb{A} = Act \times \mathbb{Z}$, $\mathbb{S} = \{c \in \mathbb{C} \mid inv(c) = \mathrm{true}\}$ and $\mathbb{S}_0 = \{c \in \mathbb{S} \mid init(c) = \mathrm{true}\}$. Also, for each $c \in \mathbb{S}$ and for each $a \in Act$ and $p \in \mathbb{Z}$ such that $R_a(c,p) = \mathrm{true}$, if $F_a(c,p) = c'$ and $inv(c') = \mathrm{true}$ 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 system invariant does not hold.

Example 3.4 (*List underlying LTS*). Given the action system described in Example 3.2, Figure 3 presents a finite fragment of its underlying LTS. List configurations are given using $[a,b,c]$ to represent the list with elements a, b and c (in that order).

Notice that, even fixing the possible elements to be 18, 32 or 25 and leaving out the `destroy` operation, the state space is infinite. Dashed lines are used to represent the extra edges that reach LTS nodes that were left out of the chosen finite fragment.

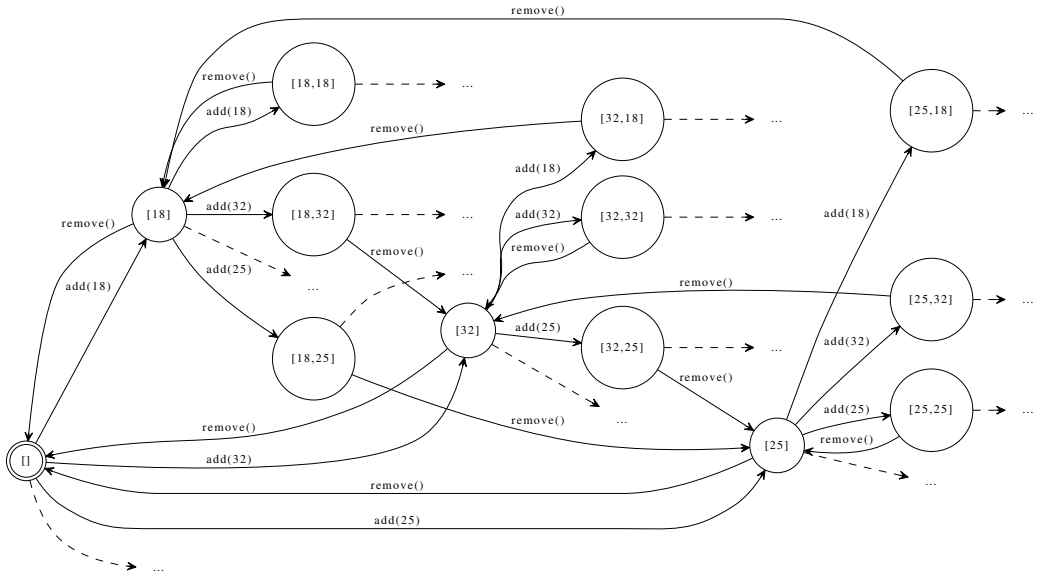

Fig. 3: Finite fragment of the list underlying LTS

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 [de Caso et al. 2010].

Definition 3.5 (*Enabledness Equivalence*). Given an action system $AS = \langle Act,\ F,\ R,\ inv,\ init \rangle$ 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 $a \in Act$ $\exists\ p \in \mathbb{Z}\ .\ R_a(c_1, p) = \text{true} \Leftrightarrow \exists\ p' \in \mathbb{Z}\ .\ R_a(c_2, p') = \text{true}$.

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

Example 3.6 (*Enabledness Equivalent List Instances*). Continuing with the list example from Figure 1, we argue that the following two instances are enabledness equivalent:

(1) A list of size 3, with nodes carrying the integers $1, 3, 5$.
(2) A list of size 2, with nodes carrying the integers $39, 10$.

This two instances enable the same set of actions, namely the `add`, `remove` and `destroy` operations.

We use a non-deterministic finite LTS to provide an abstract representation of an action system, or more precisely, of the state space defined by its infinite LTS. Simply, a non-deterministic finite LTS is a structure $M = \langle S,\ S_0,\ \Sigma,\ \delta \rangle$ where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, Σ is a finite alphabet and $\delta : S \times \Sigma \to 2^S$ is a transition function.

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 3.7 (*Enabledness-preserving Abstraction*). Given an action system $AS = \langle Act,\ F,\ R,\ inv,\ init\rangle$ 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 iff there exists a total function $\alpha : \mathbb{S} \to S$ such that $\alpha(\mathbb{S}_0) \subseteq S_0$ and for every $c \in \mathbb{S}$, action label a and parameter p such that $R_a(c,p)$ holds, then $\alpha(\Delta(c,(a,p))) \in \delta(\alpha(c),a)$. Furthermore, given a pair of configurations c_1, c_2 on $\mathbb{S}$, it holds that $c_1 \equiv_e c_2 \ \Leftrightarrow\ \alpha(c_1) = \alpha(c_2)$.

Where α is extended so that it can also be used as a function in $2^{\mathbb{S}} \to 2^{S}$ in the natural way.

Example 3.8 (*Enabledness-preserving Abstraction for List*). Figure 2 depicts an enabledness-preserving abstraction for the action system induced by the program in Figure 1.

In order to construct an enabledness-preserving abstraction we first define the notion of *action set predicate*. Given a subset of actions $A \subseteq Act$ of an action system AS, we wish to characterise all configurations c that satisfy the action system invariant inv and in which every action a in A 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 A it is *not* possible from c.

Definition 3.9 (*Predicate of an Action Set*). Given an action system $AS = \langle Act,\ F,\ R,\ inv,\ init\rangle$, the *predicate of a set of actions* $A \subseteq Act$ is the function $\mathbf{pred}_A : \mathbb{C} \to \{\mathrm{true}, \mathrm{false}\}$ defined as:

$$\mathbf{pred}_A(c) \overset{\text{def}}{\Leftrightarrow} inv(c) \ \wedge \bigwedge_{a \in A} \exists p.\ R_a(c,p) \ \wedge \bigwedge_{a \notin A} \nexists p.\ R_a(c,p)$$

Example 3.10 (*Predicate of the* $\{\texttt{add}, \texttt{destroy}\}$ *Action Set from List*). The predicate for the $\{\texttt{add}, \texttt{destroy}\}$ action set is obtained by indicating that there always exists parameters that enable `add` and `destroy`, while there is not any parameter that enables `remove`. Particularly, in the case of `destroy` and `remove` they are parameterless, so this can be simplified, obtaining:

$$\mathbf{pred}_{\{\texttt{add},\texttt{destroy}\}}(c) \ = \ inv(c) \ \wedge \ \exists p. R_{\texttt{add}}(c,p) \ \wedge \ R_{\texttt{destroy}}(c) \ \wedge \ \neg R_{\texttt{remove}}(c)$$

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 A and B with a label a when there is a configuration c satisfying the predicate of the action set A, such that when executing the action a, c evolves into a configuration that satisfies the predicate of the action set B.

THEOREM 3.11 (EPA CHARACTERISATION). Given an action system $AS = \langle Act,\ F,\ R,\ inv,\ init\rangle$, then $M = \langle \Sigma,\ S,\ S_0,\ \delta\rangle$ is an EPA of AS where:

(1) $\Sigma = Act$
(2) $S = 2^{Act}$
(3) $S_0 = \{A \in S \mid \exists c \in \mathbb{C}.\ \mathbf{pred}_A(c) \wedge init(c)\}$
(4) For all $A \in S$ and $a \in \Sigma$, if $a \notin A$ then $\delta(A,a) = \varnothing$, otherwise:

$$\delta(A,a) = \left\{ B \ \middle|\ \begin{array}{c} \exists c.\ \mathbf{pred}_A(c) \ \wedge \ \exists p.\ R_a(c,p) \\ \wedge\ \mathbf{pred}_B\big(F_a(c,p)\big) \end{array} \right\}$$

PROOF. Let $L = \langle \mathbb{A},\ \mathbb{S},\ \mathbb{S}_0,\ \Delta \rangle$ be the semantic interpretation of AS. Let $\alpha : \mathbb{S} \to S$, defined as follows:

$$\alpha(c) \stackrel{\text{def}}{=} \{a \in Act \mid \exists\, p \in \mathbb{Z} \,.\, R_a(c,p) = \text{true}\}$$

We first postulate the following lemma:

$$\alpha(c) = A \ \Leftrightarrow\ \text{pred}_A(c) = \text{true} \qquad \text{for } c \in \mathbb{S} \text{ and } A \subseteq S$$

The proof for this lemma follows directly from the definition of pred_A and α.

In order to show that M satisfying the above conditions is an EPA for AS, we have to check the following items:

i) **Initial states:**

$$\alpha(\mathbb{S}_0) \subseteq S_0$$

Remember, by Definition 3.3, $\mathbb{S} = \{c \in \mathbb{C} \mid inv(c) = \text{true}\}$ and $\mathbb{S}_0 \subseteq \mathbb{S}$ is such that $\mathbb{S}_0 = \{c \in \mathbb{S} \mid init(c) = \text{true}\}$.
Let $A \in \alpha(\mathbb{S}_0)$. Then $A \in \alpha(\{c \in \mathbb{S} \mid init(c) = \text{true}\})$. It therefore exists $c \in \mathbb{S}$ such that $init(c) = \text{true}$ and $A = \alpha(c)$.
From the lemma above, we know that it exists $c \in \mathbb{S}$ such that $\text{pred}_A(c) = \text{true}$ and $init(c) = \text{true}$. These are exactly the conditions for A to be part of the S_0 set, which is what we wanted to prove.

ii) **Transitions:** for every $c \in \mathbb{S}$, action label a and parameter p such that $R_a(c,p)$ holds, then:

$$\alpha(\Delta(c,(a,p))) \in \delta(\alpha(c), a)$$

Remember, by Definition 3.3, under these conditions, $\Delta\big(c,(a,p)\big) = F_a(c,p)$.
Let $A = \alpha(c)$ and $B = \alpha(\Delta(c,(a,p))) = \alpha(F_a(c,p))$.
We have to prove that $B \in \delta(\alpha(c), a)$. More precisely, we have to check that:

$$\exists c_0.\ \text{pred}_A(c_0)\ \wedge\ \exists p_0.\ R_a(c_0,p_0)\ \wedge\ \text{pred}_B\big(F_a(c_0,p_0)\big)$$

We claim that $c_0 = c$ and $p_0 = p$ satisfy the conditions. We analyse each conjunct:
— Since $A = \alpha(c)$, using the lemma above we know that $\text{pred}_A(c) = \text{true}$.
— Since $c_0 = c$ and $p_0 = p$, we know that $R_a(c_0,p_0) = \text{true}$.
— Using the same lemma, since $B = \alpha(F_a(c,p))$, we also get $\text{pred}_B(F_a(c,p)) = \text{true}$.

iii) **Enabledness:** for every pair of configurations c_1, c_2, then:

$$c_1 \equiv_e c_2 \ \Leftrightarrow\ \alpha(c_1) = \alpha(c_2)$$

This is directly satisfied by construction, since we defined α so that it only keeps track of the enabled actions.

□

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

Algorithm 1, presented in this section, performs a Breadth-first search (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 3.11 otherwise suggests).

ALGORITHM 1: EPA Construction

Input: An action system $AS = \langle Act,\ F,\ R,\ inv,\ init \rangle$
Output: The EPA $M = \langle \Sigma,\ S,\ S_0,\ \delta \rangle$.

$\Sigma = Act$; $S = \varnothing$;
$\delta(A, a) = \varnothing, \quad \forall A, a$;
$A^- = \{a \in Act \mid \forall c.\ init(c) \Rightarrow \neg\exists p.\ R_a(c, p)\}$;
$A^+ = \{a \in Act \mid \forall c.\ init(c) \Rightarrow \exists p.\ R_a(c, p)\}$;
$S_0^C = \{A \subseteq Act \mid A^+ \subseteq A,\ A^- \cap A = \varnothing\}$;
$S_0 = \{A \in S_0^C \mid \exists c.\ \mathbf{pred}_A(c) \wedge init(c)\}$;
$W =$ queue starting with elements in S_0;
while *there is a certain* A *at the head of* W **do**
  $W = W - [A]$;
  $S = S \cup \{A\}$;
  for *each action* $a \in A$ **do**
    $B^- = \{b \in Act \mid \forall c, p.\ \mathbf{pred}_A(c) \ \wedge\ R_a(c, p) \ \Rightarrow\ \neg\exists p'.\ R_b(F_a(c, p),\ p')\}$;
    $B^+ = \{b \in Act \mid \forall c, p.\ \mathbf{pred}_A(c) \ \wedge\ R_a(c, p) \ \Rightarrow\ \exists p'.\ R_b(F_a(c, p),\ p')\}$;
    $S^C = \{B \subseteq Act \mid B^+ \subseteq B,\ B^- \cap B = \varnothing\}$;
    for *each state* $B \in S^C$ **do**
      if $\exists c.\ \mathbf{pred}_A(c) \ \wedge\ \exists p.\ R_a(c, p) \ \wedge\ \mathbf{pred}_B(F_a(c, p))$ **then**
        $\delta(A, a) = \delta(A, a) \cup \{B\}$;
        if $B \notin S$ *and* $B \notin W$ **then**
          $W = W \cup [B]$;
        end
      end
    end
  end
end

The transition function is initialised as empty for every 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 3.11. Notice that the more actions are 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 A is visited, all of its enabled actions $a \in A$ are considered. The set B^- holds all those actions that can not be executed with any parameter after the execution of a from state A. Conversely, B^+ is the set of actions which have at least one parameter to be executed with after the execution of a from state A. The set of candidate destination states S^C is constructed in a way similar to 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 A 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 fewer candidate states the algorithm needs to consider. This optimisation makes running times come down significantly (i.e., reductions of up to 5x were observed, see Table IV in Section 5) and allowed us to cope with real-life programs while keeping time down to a few minutes in the worst case. 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 3.7.

THEOREM 4.1. Given an action system AS, M as built by Algorithm 1 is an EPA of AS.

The proof for this theorem is based on the fact that the abstraction constructed by Algorithm 1 is the reachable fragment of the abstraction presented in Theorem 3.11.

Algorithm 1 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 an action a needs to be included in the set A^-, the validity check $\forall c.\ init(c) \Rightarrow \neg\exists p.\ R_a(c,p)$ 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 6 and 16 are critical for the result of the algorithm. Line 6 affects the set of initial states; line 16 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 4.2. Let AS be an action system, and let M be built by Algorithm 1 dealing with uncertainty as follows: *a)* If uncertain in line 6 then A is added to S_0. *b)* If uncertain in line 16 then the **then**-branch is executed.

Then M satisfies a relaxation of the items in Theorem 3.11: *i)* S_0 is a superset of the one in item 3; and *ii)* $\delta(A,a)$ 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

Algorithm 1 performs logical manipulation of the mathematical functions defined in the program under analysis given by AS and therefore requires a decision engine.

Since we want 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 [de Caso et al. 2010], 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 Algorithm 1 resorting to code reachability queries.

Notice that some of the queries that we deal with in the algorithm are of the form:

$$\forall x.\ \varphi(x) \Rightarrow \psi\big(F(x)\big) \tag{1}$$

The general strategy to encode will be as follows:

---

procedure GENERAL-QUERY(x)
  if CodeOf$[\varphi]\,(x) = \text{true}$ **then**
    $y \leftarrow$ CodeOf$[F]\,(x)$
    if CodeOf$[\psi]\,(y) = \text{false}$ **then**
      TARGET

We can check if the TARGET statement is executed for at least one value of x when invoking GENERAL-QUERY(x). If TARGET is never reached, then the formula (1) holds.

In some cases the formula to analyse involves extra parameter, in some cases we need to resort to approximations, and in some other cases there are also existential quantifiers in the formula. In the rest of this section we refine this general strategy for each of the queries in the algorithm.

4.2.1. Query for line 3. For instance, given an action a, the step of line 3 requires an effective way of deciding the validity of $\forall c.\ init(c) \Rightarrow \neg\exists p.\ R_a(c,p)$. Consider the following procedure:

---

procedure a-DISABLED-ON-INIT($c : \mathbb{C},\ p : \mathbb{Z}$)
  if CodeOf$[init]\,(c) = \text{true}$ **then**
    if CodeOf$[R_a]\,(c,p) = \text{true}$ **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:

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

Meaning that the unreachability of the TARGET statement in the given procedure is equivalent to the validity of the predicate in line 3 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.

Example 4.3 (*Reachability Query for the List*). Continuing with the list example presented in Figure 1, we now show how we construct a reachability query in order to decide if the `add` action needs to be added to the A^- set.

```
procedure ADD-DISABLED-ON-INIT(l : List, e : int)
    if l = NULL ∨ (l.size = 0 ∧ l.first = NULL) then
        if l ≠ NULL then
            TARGET
```

Reachability queries like the one presented above require a decision engine to explore all posible choices for the parameters of the function at hand. In this case, a decision engine should consider all possible lists l and integer elements i and see if there is any pair (l, e) such that LIST-ADD-QUERY-FOR-LINE-3(l, e) forces the execution to visit the TARGET statement.

Notice that in this case, a non null list l with a $size$ field set to 0 and a null $first$ field makes the execution of this procedure reach the TARGET statement, therefore the `add` action is not included in the A^- set.

4.2.2. Query for line 4. 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, in line 4, we need to decide the validity of $\forall c.\ init(c) \Rightarrow \exists p.\ R_a(c, p)$. This can not be encoded as a safety property since evidence of its validity takes 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 $\widetilde{R_a}$. 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 $R_a(c, p) \Rightarrow \widetilde{R_a}(c, p)$.

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

$$\begin{aligned} \widehat{R_a}(c, p) &= \widehat{SR_a}(c) \ \wedge PR_a(p) \\ \widetilde{R_a}(c, p) &= \widetilde{SR_a}(c) \ \wedge PR_a(p) \end{aligned}$$

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 comparison queries. Usually, those could be exactly approximated by checking non-emptiness or non-nullity of substructures of configuration.

Moreover, if non-trivial candidate approximations are provided they can be verified correct. Checking the over-approximation is easy: it boils down to showing the impossibility of finding a configuration and parameter that satisfies the original clause but does not satisfy the over-approximation; which is equivalent to the TARGET statement being unreachable in the following procedure:

```
procedure a-CORRECT-OVERAPPROXIMATION(p : ℤ)
    if CodeOf [R_a] (c, p) = true then
        if CodeOf [SR̃_a] (c) = false then
            TARGET
```

The case of the underapproximation is a little bit trickier since it also requires a Skolem function Sk on the configuration for computing a candidate parameter[1]. With a Skolem function like that, checking the underapproximation boils down to verifying that Sk always finds a parameter p such that the configuration and p satisfy the original requires clause. This is equivalent to the unreachability of the TARGET statement in the following procedure:

```
procedure a-CORRECT-UNDERAPPROXIMATION( )
    if CodeOf [R_a] (c, Sk(c)) = false then
        TARGET
```

Under this scenario, the validity of the sentence in line 4 is implied by $\forall c.\ init(c) \Rightarrow \exists p.\ \widehat{R_a}(c,p)$. Since $\widehat{R_a}$ can be split in two parts, this sentence can be rewritten as $\exists p.\ PR_a(p)\ \wedge\ \forall c.\ init(c) \Rightarrow \widehat{SR_a}(c)$. 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 TARGET statement is reachable in the following procedure:

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

In fact, notice that this query does not depend on the value for the configuration c. If the TARGET statement were not reachable, then the a action can 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 $\forall c.\ init(c) \Rightarrow \widehat{SR_a}(c)$, is solved by a reachability query in this code:

```
procedure a-ENABLED-ON-INIT(c : ℂ)
    if CodeOf [init] (c) = true then
        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.

4.2.3. Query for line 6. To decide the validity of the predicates in the rest of the algorithm, given an action set $A \subseteq Act$ and a configuration c, we need to be able to determine whether $\text{pred}_A(c)$ holds. As requires clauses can be weakened and strengthened,

[1]Remember, a Skolem function value "replaces" an existentially quantified variable x in a formula φ. Its parameters are those variables in φ which are universally quantified in the scope where x appears. See [Hodges 1997] for a formal definition.

we can calculate a weaker version:

$$\widetilde{\mathbf{pred}}_A(c) \overset{\text{def}}{\Leftrightarrow} inv(c) \ \wedge \bigwedge_{a \in A} \exists p.\ PR_a(p) \ \wedge \ \widetilde{SR_a}(c) \ \wedge \bigwedge_{a \notin A} \neg\big(\exists p.\ PR_a(p) \ \wedge \ \widehat{SR_a}(c)\big)$$

This can be simplified, since $\exists p.\ PR_a(p)$ is assumed to be true. Therefore, we can calculate this approximated action set predicate using the following procedure:

---

procedure OVER-PRED-OF-A($c : \mathbb{C}$)
  $ret \leftarrow inv(c)$
  for $a \in A$ **do**
    if CodeOf$\left[\widetilde{SR_a}\right](c) =$ false **then**
      $ret \leftarrow$ false
  for $a \notin A$ **do**
    if CodeOf$\left[\widehat{SR_a}\right](c) =$ true **then**
      $ret \leftarrow$ false
  return ret

Using this action set predicate over-approximation we can decide the validity of the predicate in line 6 as a code reachability query as follows:

---

procedure A-IS-INITIAL-STATE($c : \mathbb{C}$)
  if OVER-PRED-OF-A(c) = true **then**
    if CodeOf$[init]\,(c) =$ true **then**
      TARGET

If the TARGET statement is reachable then we add A to the set S_0 of initial states. In ordet to comply with Theorem 4.2, if we are uncertain whether it is reachable or not, we still add the action set as initial state in the abstraction.

4.2.4. Query for line 12. Following a similar approximation strategy as the one used for line 3, we can now determine the validity of the check in line 12. Namely, given labels $a, b \in Act$ and an action set A, we need to decide if:

$$\forall c, p.\ \mathbf{pred}_A(c) \ \wedge \ R_a(c,p) \ \Rightarrow \ \neg\exists p'.\ R_b\big(F_a(c,p),\ p'\big)$$

In this case we will use the following logic property:

$$(\check{\varphi} \Rightarrow \hat{\psi}) \Rightarrow (\varphi \Rightarrow \psi) \qquad \text{where} \quad \varphi \Rightarrow \check{\varphi} \text{ and } \hat{\psi} \Rightarrow \psi$$

We obtain a logically weaker left-hand side of the implication:

$$\mathbf{pred}_A(c) \ \wedge \ R_a(c,p) \quad \rightsquigarrow \quad \widetilde{\mathbf{pred}}_A(c) \ \wedge \ PR_a(p)$$

And a logically stronger right-hand side:

$$\neg\exists p'.\ R_b\big(F_a(c,p),\ p'\big) \quad \rightsquigarrow \quad \neg\exists p'.\ \widehat{R_b}\big(F_a(c,p),\ p'\big)$$

The validity of which can be modeled by the following procedure:

---

procedure b-DISABLED-AFTER-a-FROM-A($c : \mathbb{C},\ p : \mathbb{Z},\ p' : \mathbb{Z}$)
  if OVER-PRED-OF-A(c) = true **then**
    if CodeOf$[PR_a]\,(p) =$ true **then**
      $c' \leftarrow$ CodeOf$[F_a]\,(c,p)$

if CodeOf $\left[\widehat{R_b}\right](c', p') = \text{true}$ **then**
  TARGET

Notice that in this case, if the TARGET statement is unreachable then every instance that satisfies a pred_A will certainly not enable b after the execution of a. On the other hand, if TARGET is reachable, then b is not necessarily enabled after the execution of a, due to the over-approximations used in the procedure. This is not a problem, since line 12 is used only as an optimisation. In other words, as we discussed earlier, not adding labels to B^- does not alter the final result of the algorithm.

4.2.5. Query for line 13. Similarly, in line 13 we need to decide the validity of:

$$\forall c, p.\ \text{pred}_A(c)\ \wedge\ R_a(c,p)\ \Rightarrow\ \exists p'.\ R_b\big(F_a(c,p),\ p'\big)$$

As in the previous case, we will use a logically weaker left-hand side of the implication by replacing:

$$\text{pred}_A(c)\ \wedge\ R_a(c,p)\quad \rightsquigarrow\quad \widetilde{\text{pred}}_A(c)\ \wedge\ PR_a(p)$$

On the right-hand side of the implication, we obtain a logically stronger formula by changing:

$$\exists p'.\ R_b\big(F_a(c,p),\ p'\big)\quad \rightsquigarrow\quad \widehat{SR_b}(F_a(c,p))$$

We construct the following procedure:

procedure b-ENABLED-AFTER-a-FROM-$A(c : \mathbb{C},\ p : \mathbb{Z})$
  if OVER-PRED-OF-$A(c)$ = true **then**
    if CodeOf $[PR_a]\,(p) = \text{true}$ **then**
      $c' \leftarrow$ CodeOf $[F_a]\,(c,p)$
      if CodeOf $\left[\widehat{SR_b}\right](c') = \text{false}$ **then**
        TARGET

The unreachability of the TARGET statement in this query will be enough evidence to indicate that b is always enabled after executing a from a configuration c which satisfies the action set predicate of A. The action b is therefore added to the B^+ set.

If TARGET is reachable, or if we are uncertain, we conservatively do not add b to B^+.

4.2.6. Query for line 16. Now we focus on the validity check in line 16:

$$\exists c.\ \text{pred}_A(c)\ \wedge\ \exists p.\ R_a(c,p)\ \wedge \text{pred}_B\big(F_a(c,p)\big)$$

In order to comply with Theorem 4.2, 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:

$$\exists c.\ \widetilde{\text{pred}}_A(c)\ \wedge\ \exists p.\ PR_a(p)\ \wedge\ \widetilde{SR_a}(c)\ \wedge \widetilde{\text{pred}}_B\big(F_a(c,p)\big)$$

Since $a \in A$, then $\widetilde{\text{pred}}_A(c)$ includes $\widetilde{SR_a}(c)$ and this sentence is equivalent to:

$$\exists c.\ \widetilde{\text{pred}}_A(c)\ \wedge\ \exists p.\ PR_a(p)\ \wedge \widetilde{\text{pred}}_B\big(F_a(c,p)\big)$$

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

```
procedure A-TO-B-USING-a(c : ℂ, p ∈ ℤ)
    if OVER-PRED-OF-A(c) = true then
        if CodeOf [PR_a] (p) = true then
            c' ← CodeOf [F_a] (c, p)
            if OVER-PRED-OF-B(c') = true then
                TARGET
```

In case of unreachability of TARGET the transition is not added to the result. If the TARGET statement is reported to be reachable, the transition is added to the result. Finally, if the decision engine is uncertain, the transition is still added to the EPA (it is suffixed with a ? symbol to report this uncertainty), therefore complying with Theorem 4.2.

4.3. Example Run of the Algorithm

In the rest of this section, we present a step-by-step execution of the Algorithm 1. We consider the action system introduced in the Example 3.2.

A^- construction:

First, we construct the A^- set of actions that are necessarily disabled in the initial state.

— `add`

```
procedure ADD-DISABLED-ON-INIT(l : List, e : int)
    if l = NULL ∨ (l.size = 0 ∧ l.first = NULL) then
        if l ≠ NULL then
            TARGET
```

} TARGET is reachable. `add` $\notin A^-$

— `remove`

```
procedure REMOVE-DISABLED-ON-INIT(l : List)
    if l = NULL ∨ (l.size = 0 ∧ l.first = NULL) then
        if l ≠ NULL ∧ l.size > 0 then
            TARGET
```

} TARGET is unreachable. `remove` $\in A^-$

— `destroy`

```
procedure DESTROY-DISABLED-ON-INIT(l : List)
    if l = NULL ∨ (l.size = 0 ∧ l.first = NULL) then
        if l ≠ NULL then
            TARGET
```

} TARGET is reachable. `destroy` $\notin A^-$

$\boxed{A^- = \{\texttt{remove}\}}$

A^+ construction:

We proceed by constructing the A^+ set of actions that are necessarily enabled in the initial state.

— `add`

```
procedure ADD-ENABLED-ON-INIT(l : List)
    if l = NULL ∨ (l.size = 0 ∧ l.first = NULL) then
        if ¬(l ≠ NULL) then
            TARGET
```

} TARGET is reachable. `add` $\notin A^+$

— `remove`

`remove` is already in A^-, it can not be in A^+ too.

— `destroy`

procedure DESTROY-ENABLED-ON-INIT(l : List)
  if l = NULL $\vee$ ($l.size = 0 \wedge l.first$ = NULL) **then**
    if $\neg(l \neq$ NULL) **then**
      TARGET

} TARGET is reachable. destroy $\notin A^+$

$$\boxed{A^+ = \varnothing}$$

S_0 construction:

Having computed A^- and A^+ we can construct the set of initial states S_0 as those action sets A that:

(1) No action of A is included in A^-.
(2) Every action in A^+ is included in A.
(3) Have at least one configuration that satisfies both the initial condition of the action system and the action set predicate of A.

First, we construct the set S_0^C of candidate states that satisfy the first 2 conditions.

$$\boxed{S_0^C = \left\{\varnothing,\ \{\texttt{add}\},\ \{\texttt{destroy}\},\ \{\texttt{add}, \texttt{destroy}\}\right\}}$$

We now test the third condition on each of the states in the S_0^C set.

— $A = \varnothing$

procedure OVER-PRED-OF-$\varnothing$(l : List)
  $ret \leftarrow l =$ NULL $\vee\ l.size \geqslant 0$ // *list invariant*
  $ret \leftarrow ret \wedge \neg(l \neq$ NULL) // *add is not enabled*
  $ret \leftarrow ret \wedge \neg(l \neq$ NULL $\wedge\ l.size > 0)$ // *remove is not enabled*
  $ret \leftarrow ret \wedge \neg(l \neq$ NULL) // *destroy is not enabled*
  return ret

procedure $\varnothing$-IS-INITIAL-STATE(l : List)
  if OVER-PRED-OF-$\varnothing$(l) **then**
    if $l =$ NULL $\vee$ ($l.size = 0 \wedge l.first =$ NULL) **then**
      TARGET

} TARGET is reachable. $\varnothing \in S_0$

— $A = \{\texttt{add}\}$

procedure OVER-PRED-OF-{add}(l : List)
  $ret \leftarrow l =$ NULL $\vee\ l.size \geqslant 0$ // *list invariant*
  $ret \leftarrow ret \wedge l \neq$ NULL // *add is enabled*
  $ret \leftarrow ret \wedge \neg(l \neq$ NULL $\wedge\ l.size > 0)$ // *remove is not enabled*
  $ret \leftarrow ret \wedge \neg(l \neq$ NULL) // *destroy is not enabled*
  return ret

procedure {add}-IS-INITIAL-STATE(l : List)
  if OVER-PRED-OF-{add}(l) **then**
    if $l =$ NULL $\vee$ ($l.size = 0 \wedge l.first =$ NULL) **then**
      TARGET

} TARGET is unreachable. {add} $\notin S_0$

— $A = \{\texttt{destroy}\}$

procedure OVER-PRED-OF-{destroy}(l : List)
  $ret \leftarrow l =$ NULL $\vee\ l.size \geqslant 0$ // *list invariant*
  $ret \leftarrow ret \wedge \neg(l \neq$ NULL) // *add is not enabled*
  $ret \leftarrow ret \wedge \neg(l \neq$ NULL $\wedge\ l.size > 0)$ // *remove is not enabled*
  $ret \leftarrow ret \wedge l \neq$ NULL // *destroy is enabled*
  return ret

procedure {destroy}-IS-INITIAL-STATE(l : List)
  if OVER-PRED-OF-{destroy}(l) **then**
    if $l =$ NULL $\vee$ ($l.size = 0 \wedge l.first =$ NULL) **then**
      TARGET

} TARGET is unreachable. {destroy} $\notin S_0$

— $A = \{\texttt{add}, \texttt{destroy}\}$

procedure OVER-PRED-OF-{add, destroy}(l : List)
  $ret \leftarrow l =$ NULL $\vee\ l.size \geqslant 0$ // *list invariant*
  $ret \leftarrow ret \wedge l \neq$ NULL // *add is enabled*
  $ret \leftarrow ret \wedge \neg(l \neq$ NULL $\wedge\ l.size > 0)$ // *remove is not enabled*

```
    ret ← ret ∧ l ≠ NULL                                   // destroy is enabled
    return ret
procedure {add, destroy}-IS-INITIAL-STATE(l : List)
    if OVER-PRED-OF-{add, destroy}(l) then
        if l = NULL ∨ (l.size = 0 ∧ l.first = NULL) then
            TARGET
```

} TARGET is reachable. $\{\texttt{add}, \texttt{destroy}\} \in S_0$

$$\boxed{S_0 = \left\{\varnothing,\ \{\texttt{add}, \texttt{destroy}\}\right\}}$$

Exploration from initial states in S_0:

Having computed the set of initial states, we initialize a work queue W that will be used in the exploration phase of the algorithm. Initially, $W = [\varnothing,\ \{\texttt{add}, \texttt{destroy}\}]$

While W is not empty, we extract the head and explore all the enabled actions.

— $A = \varnothing, \quad W = [\{\texttt{add}, \texttt{destroy}\}]$

There are no enabled actions in A to explore.

— $A = \{\texttt{add}, \texttt{destroy}\}, \quad W = [\,]$

In this case there are 2 enabled actions. We first explore the add action.

— a = add

B^- construction:

We construct the set B^- of actions that are necessarily disabled after executing add from the state $\{\texttt{add}, \texttt{destroy}\}$.

— add

```
procedure add-DISABLED-AFTER-add-FROM-{add, destroy}(l : List, e : int, e' : int)
    if OVER-PRED-OF-{add, destroy}(l) then
        l' ← add(l, e)
        if l' ≠ NULL then
            TARGET
```

} TARGET is reachable. add $\notin B^-$

— remove

```
procedure remove-DISABLED-AFTER-add-FROM-{add, destroy}(l : List, e : int)
    if OVER-PRED-OF-{add, destroy}(l) then
        l' ← add(l, e)
        if l' ≠ NULL ∧ l'.size > 0 then
            TARGET
```

} TARGET is reachable. remove $\notin B^-$

— destroy

```
procedure destroy-DISABLED-AFTER-add-FROM-{add, destroy}(l : List, e : int)
    if OVER-PRED-OF-{add, destroy}(l) then
        l' ← add(l, e)
        if l' ≠ NULL then
            TARGET
```

} TARGET is reachable. destroy $\notin B^-$

$$\boxed{B^- = \varnothing}$$

B^+ construction:

We now construct the set B^+ of actions that are necessarily enabled after executing add from the state $\{\texttt{add}, \texttt{destroy}\}$.

— add

```
procedure add-ENABLED-AFTER-add-FROM-{add, destroy}(l : List, e : int)
    if OVER-PRED-OF-{add, destroy}(l) then
        l' ← add(l, e)
        if ¬(l' ≠ NULL) then
            TARGET
```

} TARGET is reachable. add $\notin B^+$

— remove
procedure remove-ENABLED-AFTER-add-FROM-{add, destroy}(l : List, e : int)
  if OVER-PRED-OF-{add, destroy}(l) **then**
    $l' \leftarrow$ add(l, e)
    if $\neg(l' \neq$ NULL $\wedge\ l'.size > 0)$ **then**
      TARGET

} TARGET is reachable. remove $\notin B^+$

— destroy
procedure destroy-ENABLED-AFTER-add-FROM-{add, destroy}(l : List, e : int)
  if OVER-PRED-OF-{add, destroy}(l) **then**
    $l' \leftarrow$ add(l, e)
    if $l' \neq$ NULL **then**
      TARGET

} TARGET is reachable. destroy $\notin B^-$

$\boxed{B^+ = \varnothing}$

Explore candidate states:

Having computed both B^- and B^+ we can explore all the states S^C that comply with the restrictions imposed by these. In this particular case, since both sets of restrictions are empty, the set of candidate states will be complete.

$$\boxed{S^C = \left\{ \begin{array}{c} \varnothing,\ \{\texttt{add}\}\ ,\ \{\texttt{remove}\}\ ,\ \{\texttt{add}, \texttt{remove}\}\ ,\ \{\texttt{destroy}\}\ , \\ \{\texttt{add}, \texttt{destroy}\}\ , \{\texttt{remove}, \texttt{destroy}\}\ ,\ \{\texttt{add}, \texttt{remove}, \texttt{destroy}\} \end{array} \right\}}$$

We consider each candidate state B at a time, trying to determine if A can advance to B using action a. If a state is reached for the first time, it is added to the W queue.

— $B = \varnothing$
procedure {add, destroy}-TO-$\varnothing$-USING-add(l : List, e : int)
  if OVER-PRED-OF-{add, destroy}(l) **then**
    $l' \leftarrow$ add(l, e)
    if OVER-PRED-OF-$\varnothing$(l') **then**
      TARGET

} TARGET is reachable. $\varnothing \in \delta(\{\texttt{add}, \texttt{destroy}\}\ , \texttt{add})$

B is already in S, so it is not added to W.

— $B = \{\texttt{add}\}$
procedure {add, destroy}-TO-{add}-USING-add(l : List, e : int)
  if OVER-PRED-OF-{add, destroy}(l) **then**
    $l' \leftarrow$ add(l, e)
    if OVER-PRED-OF-{add}(l') **then**
      TARGET

} TARGET is unreachable. $\{\texttt{add}\} \notin \delta(\{\texttt{add}, \texttt{destroy}\}\ , \texttt{add})$

— $B = \{\texttt{remove}\}$
procedure {add, destroy}-TO-{remove}-USING-add(l : List, e : int)
  if OVER-PRED-OF-{add, destroy}(l) **then**
    $l' \leftarrow$ add(l, e)
    if OVER-PRED-OF-{remove}(l') **then**
      TARGET

} TARGET is unreachable. $\{\texttt{remove}\} \notin \delta(\{\texttt{add}, \texttt{destroy}\}\ , \texttt{add})$

$\vdots$

— $B = \{\texttt{add}, \texttt{remove}, \texttt{destroy}\}$
procedure {add, destroy}-TO-{add, remove, destroy}-USING-add(l : List, e : int)
  if OVER-PRED-OF-{add, destroy}(l) **then**
    $l' \leftarrow$ add(l, e)
    if OVER-PRED-OF-{add, remove, destroy}(l') **then**
      TARGET

} TARGET is reachable. $\{\texttt{add}, \texttt{remove}, \texttt{destroy}\} \in \delta(\{\texttt{add}, \texttt{destroy}\}\ , \texttt{add})$

$W = W \cup [\{\texttt{add}, \texttt{remove}, \texttt{destroy}\}] = [\{\texttt{add}, \texttt{remove}, \texttt{destroy}\}]$

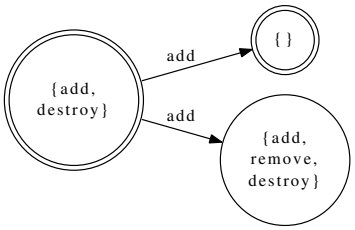

Fig. 4: Partially explored `List` EPA

At the end of this step, the algorithm has already explored the `add` action from the $\{\texttt{add}, \texttt{destroy}\}$ state. The partially explored EPA is depicted in Figure 4.

After this point, the algorithm will explore the `destroy` action from the $\{\texttt{add}, \texttt{destroy}\}$ state, and finally it will explore all the actions enabled in the $\{\texttt{add}, \texttt{remove}, \texttt{destroy}\}$ state. Due to space restrictions, we will not show a step-by-step tracing of the rest of the execution.

4.4. Implementation Details

We implemented Algorithm 1 in the CONTRACTOR tool, already presented by the authors at [de Caso et al. 2010]. This tool was originally targeted at constructing Enabledness-preserving abstractions from pre/postcondition contracts. It originally used a SMT solver (e.g. [Barrett and Berezin 2004]) in order to solve declarative logic queries. We extended this tool so that it can handle C source code via reachability queries.

Our CONTRACTOR extension takes a program equipped with requires clauses as input and produces an EPA. Internally, it follows Algorithm 1 and constructs the reachability queries presented in the previous section. At each step, a software model checker is invoked on the original program extended with a reachability procedure (QUERY-FOR-LINE-x) in order to decide if the TARGET statement is reachable.

In particular, our CONTRACTOR extension currently uses BLAST [Beyer et al. 2007] as software model checker.

BLAST input is a tuple $\langle P, l, f \rangle$ where P is a C program, l is a label defined somewhere in that program and f is the point-of-entry function to that program. Whenever we show a reachability query in Section 4.1, we define:

— f as the name of the function (e.g., b-DISABLED-AFTER-a-FROM-A),
— $l =$ TARGET,
— and P to be the program composed of the original C code that defines AS extended with the function f.

Given a tuple $\langle P, l, f \rangle$, BLAST tries to find an instantiation for every parameter of f such that the execution of f using those parameters reaches l in program P. As we mentioned before, reachability solving is undecidable in general so BLAST may not be successful at finding a parameter valuation that hits l even when it exists. In any case, Theorem 4.2 guarantees that our result is a safe overapproximation.

The exploration for concrete parameter values is tricker when f has a formal parameter of a non-primitive type τ (e.g., a C `struct`). In such scenarios, τ instances need to comply with an internal invariant I_τ. To the best of our knowledge, there is no explicit mechanism in BLAST to impose an invariant on a complex type. Instead, if an action a takes a parameter p of type τ, we add $I_\tau(p)$ to the requires clause R_a. Parameters that do not comply with R_a are not considered by the reachability queries, therefore avoiding malformed instances of τ as witnesses for reachability. Failing to include I_τ in R_a

could imply having extra transitions in the resulting EPA, which does not compromise its soundness.

Regarding our tool architecture, we use BLAST as a black-box component[2]. Doing so may reduce our chances to incorporate optimisations in the EPA construction process. On the other hand, using BLAST as a black-box enables CONTRACTOR to be defined in a modular style that would allow BLAST to be replaced with other back-ends if needed.

For instance, instead of using a software model checker, we could have used a verification-based approach (e.g., [Cok and Kiniry 2005]). As we further mention in Section 6, we explored such possibility in [Zoppi et al. 2011].

Another option would be to use a symbolic execution engine (e.g., [Khurshid et al. 2003]). However, most symbolic execution engines fail to capture the complete behaviour of a program (e.g., due to loop unrolling). In this scenario, it is harder to guarantee that EPAs indeed exhibit an overapproximation of the behaviour.

Finally, an alternative approach to solving reachability queries would be to use a testing-based approach. For each query, we could use a random test-case generator (e.g., RANDOOP [Pacheco and Ernst 2007]) for a limited number of time. If at least one of the test-cases reaches the TARGET statement, then we would add the transition. If none of the test-cases hits the TARGET statement, then there is no guarantee. Instead of obtaining a behaviour overapproximation, our EPA would feature only a subset of the legal behaviour, possibly difficulting the user validation process.

4.5. About the Technique's Assumptions

In this section and the previous one we presented results that rely on two assumptions. We assumed user-provided invariants to be accurate. We also assume that requires clauses can be split. In this section we discuss the impact of violating these assumptions.

4.5.1. Violating the Invariant Correctness Assumption. We split the discussion in two scenarios:

a) First, we consider user-provided invariants that are too weak. This means that the user provided invariant admits instances that are not reachable using the provided set of actions. In such cases, as a direct consequence the abstract state predicates become weaker than they should. Therefore, when constructing transitions, we will possibly consider concrete class instances that satisfy the supplied (weak) invariant. As a consequence, extra transitions could appear since they would use these bogus concrete intances as witnesses.
   In this case, the constructed EPA will still be an overapproximated version of the class behaviour. However, in extreme cases (for instance, when the invariant is set to true) the resulting EPA could be very different from the one we would get with a more accurate invariant. This abrupt different with respect to the expected result (i.e., one that matches the mental model) can be a hint for the developer that she needs to provide a refined version of the invariant.
b) Second, the user can provide an incorrect invariant. That is, one that is falsified by at least one legal instance of the class. By *legal instance* we refer to an instance that can be constructed by starting from a valuation satisfying the initial predicate $init$ and arbitrarily invoking any number of actions in A.
   For such cases, our current version of CONTRACTOR provides an experimental feature that checks the validity of the user-provided invariant on each transition. Extra reachability queries similar to the ones presented in this section are used for

[2]It is worth mentioning that BLAST uses predicate abstraction internally, but this is independent from our general abstraction approach since we use this tool as a black box.

this purpose. The offending transitions are then marked with a $*$ in the output, so that the user can realize that there is a problem with the invariant, and what action triggers it. The details of this experimental feature are not further presented in this work.

4.5.2. Violating the Requires Clauses Splitting Assumption. Regarding the requires clauses splitting problem, based on our observations in a number of industrial APIs, we present 3 common patterns of requires clauses. In the following, let $x_1, \ldots, x_k$ be a subset of the parameters and let $y_1, \ldots, y_m$ be a subset of the API internal variables (or fields).

— $P_1(x_1, \ldots, x_k) \wedge P_2(y_1, \ldots, y_m)$
An example of this is a `push` operation for a stack that stores possitive numbers. The requires clause should check that the element being pushed is not negative (P_1) and that the stack is not full (P_2). Splitting this kind of requires clauses is trivial since we set PR as P_1 and SR as P_2.

— $P(y_1, \ldots, y_m)$
This is the case for many actions that take no parameters. An example of this is the `close` operation for a file handler. The only requirement is that the file is open, and there are no parameters. This pattern also appears when the action takes parameters, but imposes no restriction on them, such as data containers. Splitting this kind of requires clauses is also trivial, as it is a particular case of the previous one where P_1 is set to true.

— $P_1(x_1, \ldots, x_k) \wedge P_2(y_1, \ldots, y_m) \wedge x_i \ \mathrm{op} \ y_j$
This third pattern adds an extra check that involves a comparison of a field variable and a parameter. An example of this is a `login` operation that checks that the given password (as provided by the user in the parameter) matches the password that is stored in a field. In this example op is the equality operation.
There is no generalised way to split this kind of requires clauses. However, for the purposes of EPA construction, requires clauses are used to determine whether actions are enabled or not. Therefore, in such cases, the existential elimination of the parameter x_i can yield a reasonable approximation of the requires clause. Earlier in Section 4.2.2 we do provide reachability queries that check if the provided requires clause approximation is sound.
For instance, in the `login` example, from an enabledness point of view, the password check is irrelevant. In other words, there is always the possibility that the user will provide the correct password, therefore the check that the input password matches the stored password can be dropped.
A similar example arises when the requires clause for an action specifies that a parameter value has to be part of a collection stored in a field. For instance, a `process` operation that takes the key k of an active work item and checks that k belongs to a stored list of active work items W. In this example, op is the "belongs" $\in$ set operation. From an enabledness perspective, as long as the active work items set W is not empty, there will always exist a key k that will enable the `process` action. Therefore, the $k \in W$ restriction can be relaxed and rewritten as $|W| > 0$.

There are other patterns, such as multiple parameters x_{i_1}, x_{i_2} being compared with several fields $y_{j_1}, y_{j_2}, y_{j_3}$. However, we did not find this kind of situations in practice.

4.5.3. About Requires Clauses Correctness. Notice that, even though we need requires clauses to be splittable, the technique does not require any notion of requires clause correctness. Furthermore, there is no general way to define what requires clauses should describe. In some cases, requires clauses are set so that no exceptions are thrown (this is the case in all of the classes evaluated in the next section). In some

other cases, the API uses error codes in the result (e.g., `pop` returns -1 if there are no elements) and requires clauses have to be defined so that these special values are avoided. Finally, some other APIs are more permissive and fail silently (e.g., `push` leaves the stack as is if there is no more room for the new element).

The concept of requires clause is associated to weakest preconditions [Dijkstra 1975]. We analyse what happens when an action a has a requires clauses that does not imply the weakest precondition of F_a.

— F_a **could not terminate or terminate abnormally**. In this scenario, the underlying reachability solver (e.g., BLAST) could possibly not yield a definite answer. As we discussed earlier, this does not affect the soundness of the construction algorithm.
— F_a **terminates normally but leaves the system in a possibly inconsistent state**. In this scenario the resulting EPA may present extra a-labeled transitions. The developer could potentially discover these transitions, and then fix the problems in R_a.

Unlike invariants, requires clauses are not assumed to be accurate. There is no assumption on requires clauses but the fact that they are satisfiable, which we check via validity queries.

There are no other requirements for requires clauses. Furthermore, there is no easy criterion to determine whether a requires clause is correct or not. Requires clauses depend on the API designer's intent.

For instance, consider the case of an ATM with an action `withdraw(int amount)` with an empty requires clause. A reviewer might argue that the requires clause is incorrect since there are two conditions that need to be checked: *i)* that the given `amount` is positive, and *ii)* that the client has enough funds. While the first condition seems reasonable for all possible ATMs, the second one might not be desirable if the bank allowed clients to have a (bounded) negative balance. In other words, changing the requires clause affects the resulting functionality of the action system. Therefore, this is a validation problem, since we are comparing the behaviour of the `withdraw` action with our mental model of what this operation should do.

Since this is a validation problem, not only we do not need requires clauses to be accurate, but we also can help detect problems in them. In [de Caso et al. 2010] we describe a series of validation guidelines that help the user navigate EPAs and identify cases on which requires clauses are too weak or too strong. This is in the context of pre/postcondition contracts, but the guidelines still apply for the case of EPAs produced from source code.

In the next section we analyse our construction algorithm and its implementation on a series of industrial programs. When dealing with programs that do not explicitly mark requires clauses, a number of extra challenges arise. We discuss these in the next section.

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 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 qualita-

Name	# Actions	Non-whitespace nor comment LOC		
		API functionality	Requires clauses	Invariant
`List`	3	75	8	1
`PipedOutputStream`	4	90	10	1
`Signature`	5	83	12	1
`ListItr`	5	130	26	6
`Socket`	8	230	25	11
`PCCRR`	12	251	39	6
`SMTPServer`	9	85	19	4
`SMTPProtocol`	16	510	34	1

Table I: Case studies subjects size information

tive 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, leveraging the binding that action set predicates create between EPA transitions and states with code fragments such as requires clauses, each finding is manually 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`, `ListItr` and `Socket` from the Java Development Kit (JDK) 1.4 implementation; the `SMTPServer` server-side from the JES Java mail server; the `SMTPProtocol` client-side class from the RISTRETTO protocol-level Java mail client; and the `PCCRR` class was taken from a C# SpecExplorer protocol model. Table I presents some additional information regarding the size of the subject APIs.

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., [Alur et al. 2005; Henzinger et al. 2005]). And *iv)* classes that are of industrial relevance.

The nature of the BLAST tool, which deals with C code, forced us to analyse programs that are written in that programming language. Since most of previous work in our area focuses on analysing Java classes, we had to manually translate these to C in order to be able to compare our approach.

We used the existing run-time checks on each class source code as requires clauses. Table II presents the information regarding requires clauses splitting. Almost all the requires clauses could be split in most of the analised APIs. The exception was the `PCCRR` class: a few actions had requires clauses which forced the value of a parameter to be exactly the same as 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 $\check{R}$ to true could

Name	Total actions	Actions with precise splitting	Actions that required approximation
List	3	3	0
PipedOutputStream	4	4	0
Signature	5	5	0
ListItr	5	5	0
Socket	8	8	0
PCCRR	12	9	3
SMTPServer	9	9	0
SMTPProtocol	16	16	0

Table II: Case studies subjects requires clauses splitting information

be used as an exact approximation of the original requires clause from an enabledness point of view.

With respect to the time it took to produce the requires clauses, it is worth noticing that they were not produced from scratch, but rather identified in the existing code. While time was not accounted for the requires clauses extraction, we believe that automating this task is key to lowering the adoption barrier for CONTRACTOR.

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. [Dallmeier et al. 2010]. The ListItr EPA was compared against a manually-generated model, which was constructed by a senior Java developer. The Socket EPA was compared against the class Javadoc and an inferred restriction reported in [Henzinger et al. 2005]. The PCCRR EPA was compared against the reviewer's understanding of the protocol since the C# SpecExplorer model was undocumented. The SMTPServer and SMTPProtocol EPAs were compared against a manually-generated model made available by Dallmeier et al. [Dallmeier et al. 2010], as well as the SMTP Protocol RFC[3].

Finally, there were a few cases in which action parameters had non-primitive types (i.e., types other than bool or int). There are no a-priori limitations in our approach with respect to non-primitive parameters. However, in our current implementation we inherit the limitations of the BLAST back-end. As we will present later on this section, even when BLAST does not necessarily specialize in finding values for complex data types, it has performed reasonably well in all the scenarios that involved finding such values.

5.3. Findings

In this section we report on the most relevant findings discovered while performing the case studies. A more extensive report of these case studies, together with all the generated models is included in 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 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(PipedInputStream snk) connects the PipedOutputStream to the reader side.

[3]http://www.faqs.org/rfcs/rfc821.html

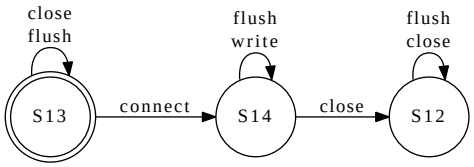

Fig. 5: EPA of JDK 1.4 `PipedOutputStream`

— `write(byte b)` outputs the given byte and makes it available to the reader side.
— `flush()` notifies the reader side of the data availability in the pipe.
— `close()` ends the connection with the reader side.

We produced a straightforward translation of the JDK 1.4 implementation of the `PipedOutputStream` to C. For the requires clauses we manually extracted the code fragments which contain the necessary conditions to avoid exceptions being thrown when executing methods of the class. It was trivial to split these requires clauses into two parts, as required by our approach, because none of them depended both on parameters and class attributes. Finally, we used true as system invariant.

The model in Figure 5 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 5 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 expert reviewer analysed if this trace was legal in two ways: *i)* by exercising this trace to see if it threw an exception; and *ii)* by analysing 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.

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, or a private key, which initializes for signing; *ii)* Updating, which updates the bytes to be signed or verified; and *iii)* Signing or verifying a signature on all updated bytes.

— `initSign(PrivateKey privateKey)` initializes the `Signature` object in signing mode. The data to be signed is initialized to an empty `byte` array.
— `initVerify(PublicKey publicKey)` initializes the `Signature` object in signature verification mode.
— `update(byte[] b)` updates the data to be signed.
— `sign()` returns a cryptographic signature for the data given by the last `update` command, if any.

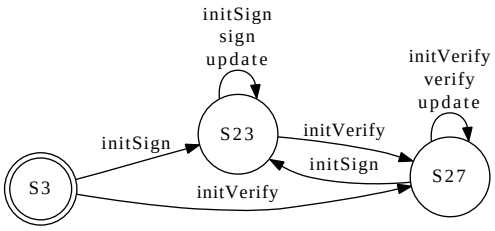

Fig. 6: EPA of JDK 1.4 `Signature`

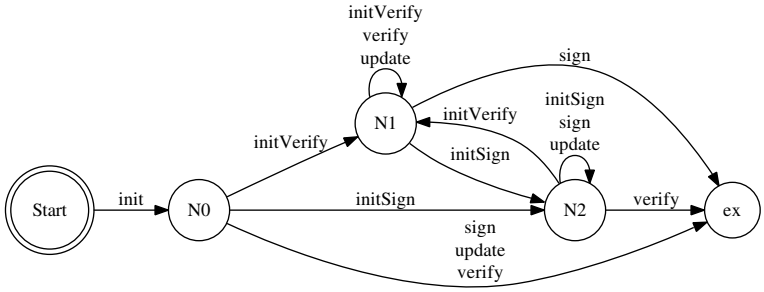

Fig. 7: Manually generated model of JDK 1.4 `Signature` (extracted from [Dallmeier et al. 2010])

— `verify(byte[] signature)` checks the cryptographic signature given in the parameter.

As with the previous class, we translated the JDK 1.4 implementation of the `Signature` to C. Similarly, we defined the requires clauses with the adequate manually extracted code fragments. We defined the system invariant as true.

The model in Figure 6 is the EPA for `Signature` obtained with CONTRACTOR on a C version of the JDK 1.4 implementation of class `Signature`, introducing split requires clauses that prevented exceptions from being thrown.

The EPA obtained with CONTRACTOR was exactly the same as the manual model presented in [Dallmeier et al. 2010]. This model clearly represents how an instance of `Signature` can only be in 3 different states: uninitialized, initialized for signing or initialized for signature verification. 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 faithful representation of both the manually-generated model presented in [Dallmeier et al. 2010] (see Figure 7), 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. The available actions on a `ListItr` object are:

— `next()` retrieves the following element, unless the end of the list has been reached.
— `prev()` retrieves the previous element, unless the iteratior points to the beginning of the list.
— `add(object o)` inserts an new element in the current position.

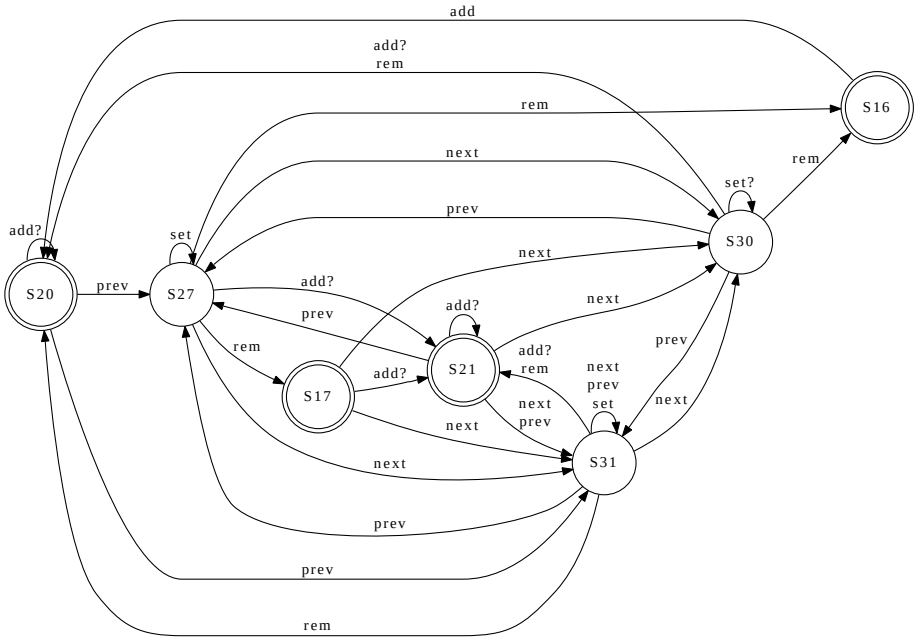

Fig. 8: EPA of JDK 1.4 `ListItr`

— `rem()` removes the last retrieved element. Therefore, it is enabled only after `next()` or `prev()` have been invoked.
— `set(object o)` replaces the last retrieved element for the given `o`. Just like `rem()`, it is only enabled after the execution of `next()` or `prev()`.

We translated the JDK 1.4 Java List Iterator implementation ranging over an `ArrayList`. The requires clauses where defined using manually extracted fragments of code so that no exception was thrown. The system invariant includes restrictions such as:

— The current size of the `ArrayList` does not exceed its capacity.
— The cursor used by the iterator is in the range of the array.
— The last returned element by the iterator is either: *a)* undefined; or *b)* next to the cursor.

The abstraction of Figure 8 is the EPA obtained by CONTRACTOR. 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 has 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.

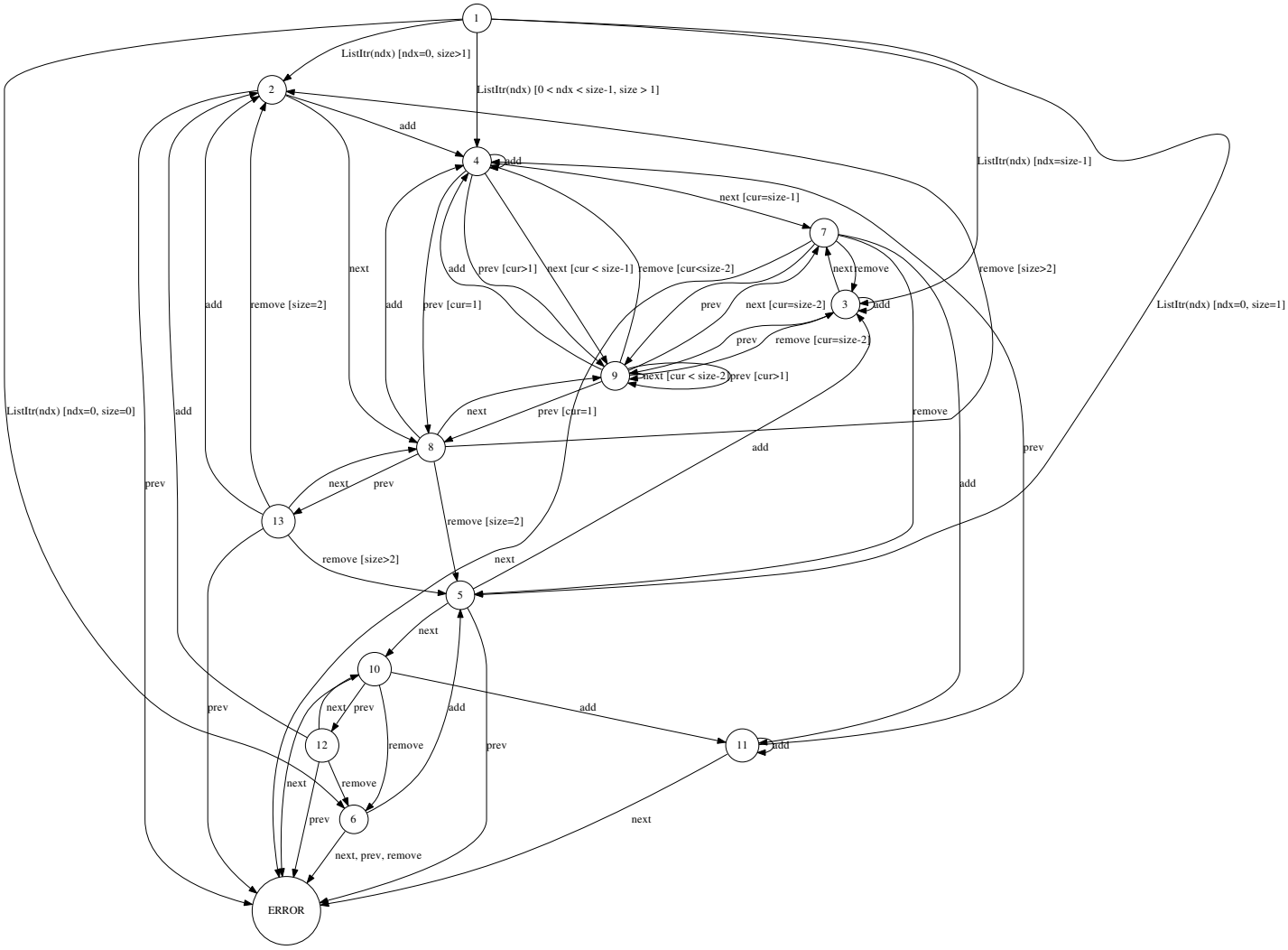

Fig. 9: Manually generated `ListItr` behaviour model

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 system invariant may not be preserved. A finer-grained manual analysis revealed that the system invariant is not violated, however BLAST is not able to prove this.

A senior Java applications developer with more than 8 years of experience (including experience with formal models) manually generated a behaviour model, shown in Figure 9, 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.

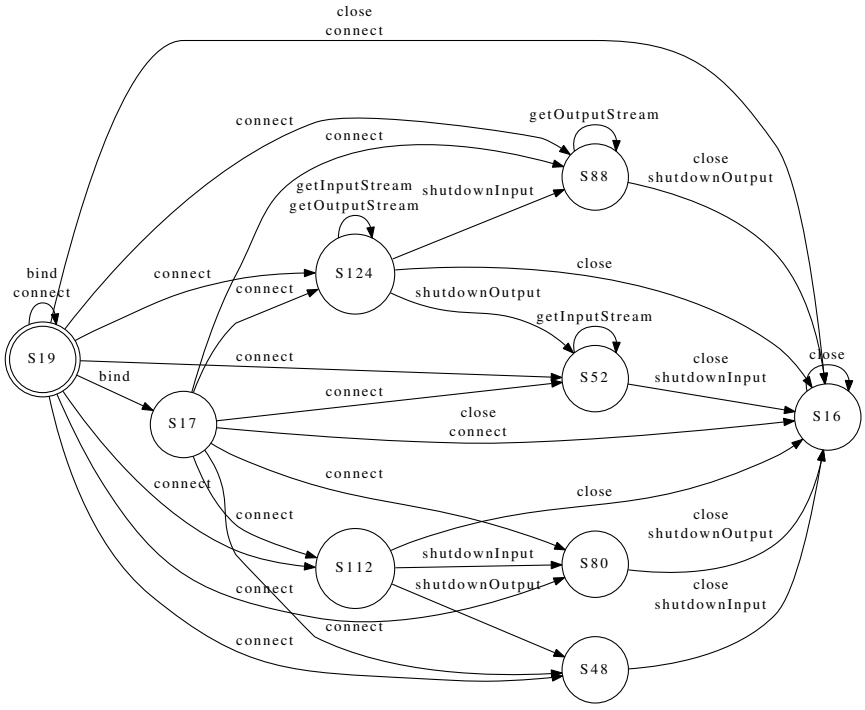

Fig. 10: First EPA of JDK 1.4 `Socket`

When comparing this manually-generated model with the EPA, an expert reviewer (which was not the same person as the developer who manually created the model) discovered that the overall level of abstraction of the manually-constructed model 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 these states, the expert reviewer discovered that they were exhibiting spurious behaviour and were accidentally introduced by the developer, due to his misunderstanding of the requirements.

5.3.4. Java Socket. A Java `Socket` provides the client-side functionality to establish a TCP connection between two hosts. A `Socket` can engage in the following actions:

— `bind(SocketAddress bindpoint)` establishes the local address (particularly, local port) of the client socket.
— `connect(SocketAddress endpoint, int timeout)` establishes a connection with a remote server socket.
— `getOutputStream()` and `getInputStream()` return the streams on which the client can send and receive from the server, respectively.
— `shutdownOutput()` and `shutdownInput()` close the sending and receiving streams, respectively.
— `close()` ends the connection with the server.

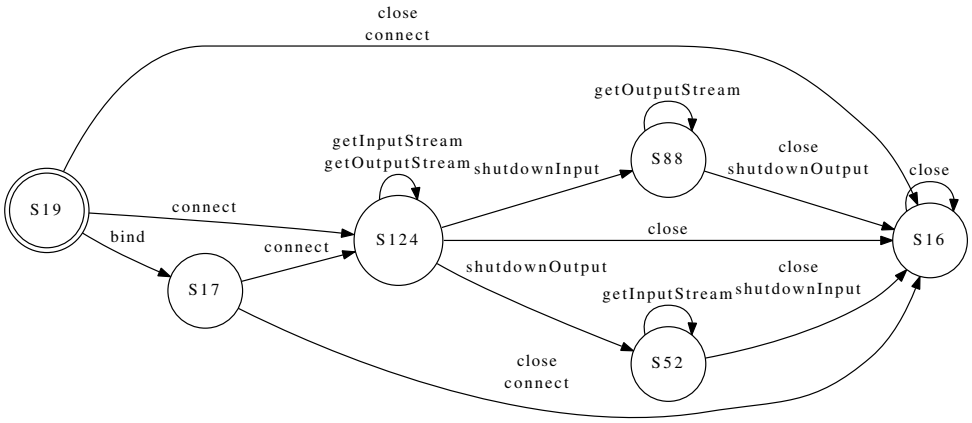

Fig. 11: Final EPA of JDK 1.4 `Socket`

We ran CONTRACTOR on a C translated version of the JDK 1.4 `Socket` implementation. The system invariant restricts that the port value is in range (from 0 to 65535), and that the fields marking the shutdown state of each stream (either input or output) actually reflect the state of the streams. Finally, we obtained the EPA depicted in Figure 10.

This first abstraction provides evidence on how the `bind` operation may be omitted before a call to `connect`, since they both eventually lead to the state `S124`. This is not clearly stated in the Java official documentation for this class.

Furthermore, this EPA shows 2 suspicious elements:

— The `connect` operation from the initial state advances to several different states, some of which do not allow sending to and/or receiving from the server. This is not the expected behaviour, since a fresh connection should not block any of these actions.
— Furthermore, some states enable the `shutdownInput` action, even when `getInputStream` is disabled. A similar situation happens with `shutdownOutput` and `getOutputStream`.

A closer inspection of the `Socket` class reveals that both of these problems were due to a weak action set predicate of the `S17` and `S19` states. The boolean variables that store whether the sending and receiving streams are closed are always `false` since the `Socket` creation. However, this restriction is not valid in all of the `Socket` states, particularly after either `shutdownInput` or `shutdownOutput` have been executed.

In order to deal with this *pseudoinvariant* which holds on some of the abstract states, we added a feature in CONTRACTOR which allows the user to specify properties which are specific to some of the abstract states.

We then added a restriction that encodes that the variables that keep track of the sending and receiving streams are closed in the `S19` and `S17` abstract states. Running CONTRACTOR on this new version produces the EPA in Figure 11.

This second abstraction shows how once the connection is established both sending and receiving are enabled. Each of these actions is disabled after its corresponding shut-down operation. The same restriction is obtained by [Henzinger et al. 2005], but it requires the user to add six predicates to keep track of the state.

5.3.5. PCCR Framework. The *Peer Content Caching and Retrieval* (PCCR)[4] system is a P2P-based distribution framework designed to reduce bandwidth consumption in wide area networks. The key feature is that it 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 [Grieskamp et al. 2011], an expert 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, 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(SutPlatform sutPlatform)` establishes which operating system runs on the client to be tested.
— `InitSut(bool isTestingNegotiation)` initializes the client to be tested, providing a parameter that indicates if the negotiation phase of the SUT is being 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(bool isTimerExpire, bool isSameSegment, bool isWellFormed, bool isOverlap)` makes the server send the hashes for the requested blocks it possesses to the client.
— `SndBlkListAb(bool isTimerExpire, bool isSameSegment, bool isWellFormed, bool isOverlap)` is the same as the previous one, but with a response indicating the request was not consistent.
— `RcvGetBlk(uint index)` indicates that the server has received a request for a particular block, indicated by its hash.
— `SndBlk(ContentType cType, bool isTimerExpire, bool isSameSegment, bool isWellFormed, uint index)` 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 restrictions between the requests that come from the client once the connection has been established. 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:

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

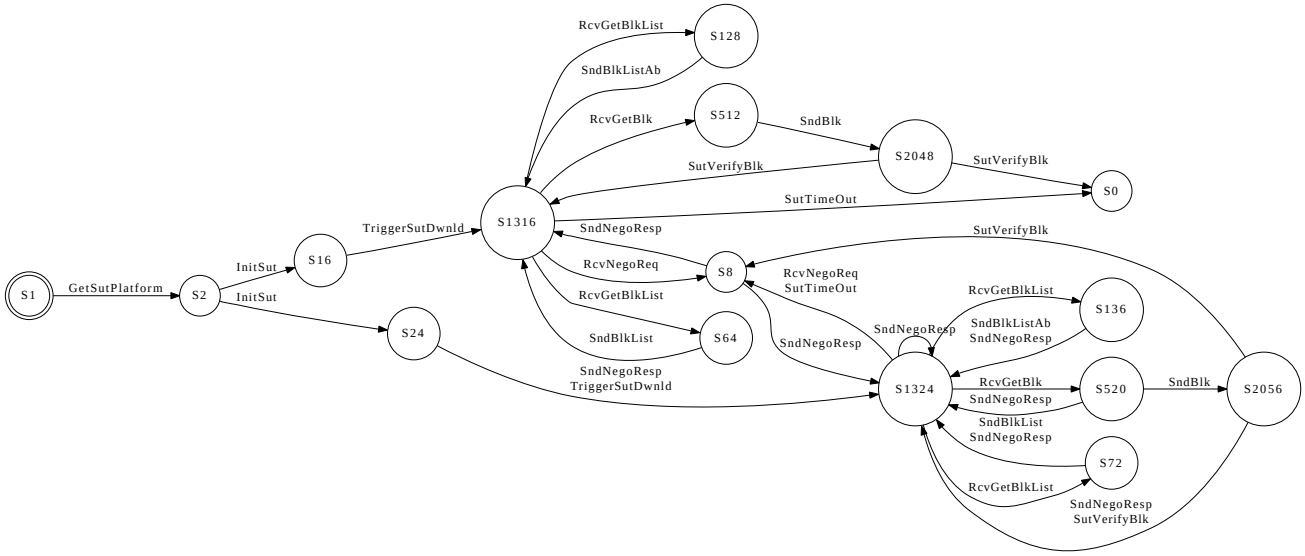

Fig. 12: First EPA of the `MS-PCCRR` server side

— `TriggerSutDwnld(ContentType cType)` makes the client under test request a download of the given kind to the server.
— `SutTimeOut()` indicates that the client has timed out.
— `SutVerifyBlk(ContentType cType, uint index)` 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. Some of the requires clauses in the original model had constraints over parameters and model variables, however these could be simplified by asumming worst-case fixed values for fields. For instance $p \leqslant f$ where p is parameter and f is field can be safely dropped since there is always going to exist such a p (in particular $p = f$ satisfies the constraint). In order to keep code safe, the $p \leqslant f$ restriction was then moved to the actual action code. The relaxed preconditions were checked correct using the approach described in Section 4.

The original C# class had a series of `enum` fields. The C version turns each `enum` field into an `int` field. The system invariant for the `PCCRR` class imposes restrictions so that these `int` fields are actually in range.

The first EPA obtained with CONTRACTOR, which had 16 states, which can be seen in Figure 12, 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, as it can evolve to states `S16` and `S24`.

The expert reviewer checked the code for the cause of non-determinism on `InitSut` and discovered that it is an action with a single boolean parameter called `isTestingNegotiation`, which is stored in a boolean field named `isTestingNego`. The reviewer then searched for other appearances of the `isTestingNego` field and found that when it this true then a negotiation response the `SndNegoResp` action is enabled.

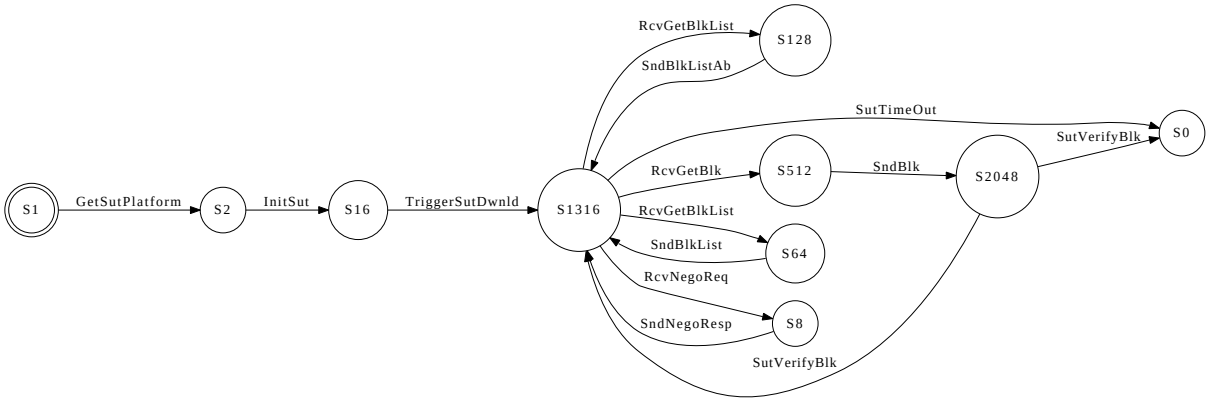

Fig. 13: Second EPA of the MS-PCCRR server side (eliminated the isTestingNego field)

In the EPA depicted in Figure 12, this issue is manifested as the 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 these sets.

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 13.

CONTRACTOR was then ran over the 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.

5.3.6. SMTP Server. The SMTPServer class is a Java implementation of an SMTP protocol server extracted from Java Email Server (JES) [5].

— ehlo(string hostname) is used by the client to indicate that it wishes to use the extended SMTP protocol. The client hostname is provided, so that the SMTP server can decided if it will relay e-mail for that domain.
— mail() indicates that the client wishes to send a new e-mail.
— rcpt(Address a) is invoked for each of the recipients that the client needs to add as recipients for the newly created e-mail.
— data(byte[] data) is used to indicate the actual contents of the e-mail. It can be invoked when at least one recipient was already provided.
— verify(Address a) is used to check if a given e-mail address is served by this SMTP server.
— rset() is a clean-up operation to restore the server back to the initial state.
— noop() is an empty command used to keep alive the connection.

[5] http://www.ericdaugherty.com/java/mailserver

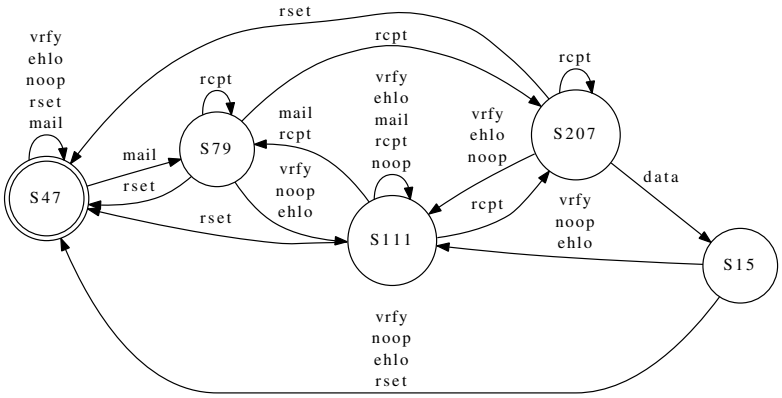

Fig. 14: EPA for SMTP protocol server class

We translated the `SMTPServer` class to C and run CONTRACTOR, using the requires clauses as extracted from the runtime checks on each method. As with the previous class, the system invariant imposes a restriction over a translated `enum` field so that it is in range. We then obtained the EPA in Figure 14.

We found two anomalies in the EPA:

(1) After a `mail` command (`S47` → `S79`), a `noop` operation modifies the SMTP server internal state (`S79` → `S111`). Particularly, instead of just being able to add new recipients, in `S111` the `mail` operation is also enabled (as many others). This non-empty behaviour for the `noop` operation is against the SMTP protocol standard.
(2) After a `data` command (`S207` → `S15`), we should be able to send a new email using the `mail` command. However, the EPA shows that this implementation requires the client to first call another command such as `noop` or `rset` in order to go back to the initial state.

Taking a closer look at the `STMPServer` source code, we found that it uses a variable to store the name of the last invoked command. Storing `noop` as the last invoked command is clearly a bad implementation strategy, since the server loses track of whichever command was executed before that. This problem was causing the first problem described above.

On top of this, the second problem is caused by an omission in the requires clause of the `mail` command, which causes the operation to remain disabled when `data` is the last executed command.

5.3.7. SMTP Client. The `SMTPProtocol` class is a Java implementation of an SMTP protocol client extracted from the RISTRETTO Java mail client[6]. The requires clauses were set according to the run-time checks found in the first lines of each method. The system invariant imposes a restriction on a translated `enum` field. CONTRACTOR was run and outputted the enabledness-preserving abstraction in Figure 15.

When compared to the manually-generated model in [Dallmeier et al. 2010] (see Figure 16) the reviewer discovered that the constructed EPA was much 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`).

[6] `http://ostatic.com/ristretto`

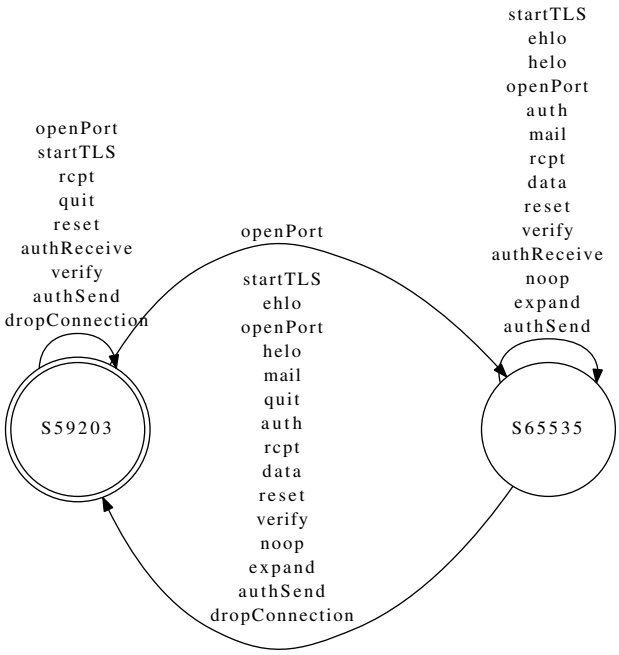

Fig. 15: EPA for SMTP protocol client class

On the other hand, the EPA does not impose ordering restrictions to any command, as long as the connection with the server is established. This lack of restrictions in the EPA is caused by the fact that the `SMTPProtocol` implementation 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.

The manually-generated model was indeed constructed by considering the behaviour that emerges when connecting the `SMTPProtocol` instance to a well behaving SMTP server (i.e., a server that complies with the ESMTP standard [Klensin et al. 1995]). Should the client connect to an SMTP server that does not follow the protocol standard, then the behaviour would significantly differ.

The pass-through behaviour that the EPA unveils can quickly help an expert reviewer realize that the `SMTPProtocol` implementation does indeed have a design flaw, since it heavily relies on the server being correctly implemented. This dependence is not reflected in the manually-generated model.

5.4. Quantitative Analysis

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

Table III 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`.

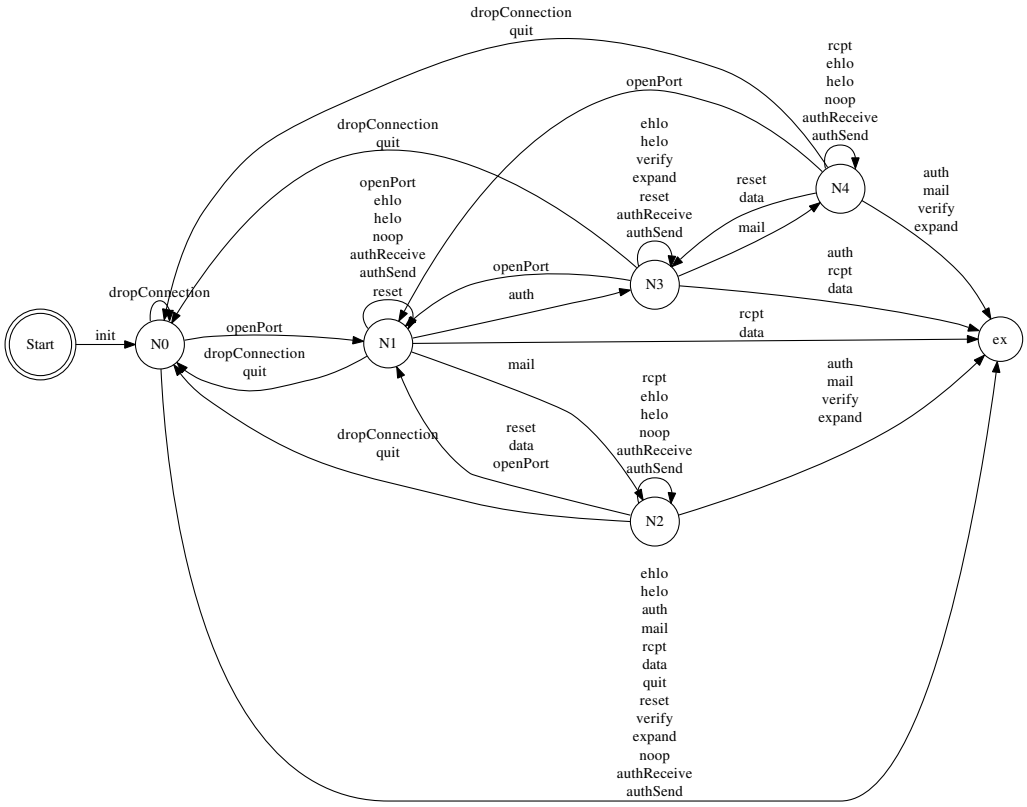

Fig. 16: Manually generated model for SMTP protocol client class (extracted from [Dallmeier et al. 2010])

Input	Tool execution		Output
#Actions, Name	**#BLAST queries (certainty)**	**Time**	**States, transitions**
3, `List` (buggy)	60 (100%)	6s.	3, 7
3, `List` (fixed)	62 (100%)	7s.	3, 8
4, `PipedOutputStream`	85 (100%)	8s.	3, 8
5, `Signature`	127 (100%)	13s.	3, 10
5, `ListItr`	279 (97.5%)	5m. 14s.	7, 32
8, `Socket` (1^{st})	400 (100 %)	42m. 11s.	9, 38
8, `Socket` (2^{nd})	255 (100 %)	24m. 32s.	6, 19
12, `PCCRR` (1^{st})	558 (100%)	10m. 28s.	16, 34
12, `PCCRR` (2^{nd})	253 (100%)	6m. 38s.	10, 14
9, `SMTPServer`	461 (99.8 %)	27m. 22s.	5, 34
16, `SMTPProtocol`	979 (100%)	5m. 18s.	2, 39

Table III: Case studies summary

It is worth mentioning that the reachable fragments of the EPAs constructed with CONTRACTOR feature significantly fewer states than the complete $2^{|Act|}$ 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.

In particular, we analysed if the overhead of computing A^+/A^- is compensated later when less actions have to be considered. Table IV 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. The exception is the `SMTPServer` class, for which the unoptimized version runs a little bit faster. This behaviour clearly reflects the fact that computing the A^+/A^- sets is linear, and is therefore amortized.

Finally, the engine certainty was very high in all of the analysed case studies, and completely certain in 4 out of 6 cases. This is remarkably high for the BLAST tool, specially considering that most of the analysed classes were relevant programs already studied in previous work [Alur et al. 2005; Henzinger et al. 2005; Dallmeier et al. 2010].

5.5. Qualitative Analysis

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

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 `Socket` case study, our abstraction was able to convey a simple, yet representative picture of the implementation state space; identifying the key parts that keep

Input	Tool execution	
Name	**Optimised running time**	**Unoptimised running time**
`List` (buggy)	6s.	4s.
`List` (fixed)	7s.	4s.
`PipedOutputStream`	8s.	6s.
`Signature`	13s.	9s.
`ListItr`	5m. 14s.	6m. 9s.
`Socket` (first)	42m 11s.	87m. 45s.
`Socket` (second)	24m. 32s.	51m. 24s.
`PCCRR` (first)	10m. 28s.	17m. 58s.
`PCCRR` (second)	6m. 38s.	11m. 38s.
`SMTPServer`	27m. 22s.	25m. 44s.
`SMTPProtocol`	5m. 18s.	28m. 10s.

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

track of the sending and receiving streams' state. It also was able to reproduce the call order restrictions obtained using previous techniques.

The `SMTPServer` case study shows how our abstraction can reveal implementation strategy errors and omissions; which, in turn, show how even implementations for popular and well understood protocols such as SMTP can be tricky to get right.

In the `SMTPProtocol` case study, our abstraction clearly reflects the fact that the client-side implementation relies 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 a few spurious transitions due to undecidability and 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.

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 relatively small (a sample of 7 programs). For instance, we have translated and analysed 2 of the 3 complete models presented in [Dallmeier et al. 2010].

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. 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 be revealed by the chosen granularity of action labels. It is possible to use labels to denote request/response pairs by providing requires clauses that ensure the expected type of response is yielded. Such a denotation would produce 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, the manually annotated requires clauses and invariants or the manually-generated models may be incorrect.

Regarding the manually-generated models, we minimise this risk by using material previously used by other authors, as well as making available all the new material. Regarding the requires clauses, their extraction followed a principled approach based on discovering exceptions thrown at the beginning of each method. Finally, while having good invariants poses a real challenge for the approach, most of the invariants we

used were obtained by formalizing colloquial descriptions found in the classes Javadoc, as well as some other type restrictions imposed by the manual translation to C.

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 this section we compare our approach to the construction of behaviour models with other previously published techniques. Table V presents a summary comparison.

From the comparison presented in this table, we can conclude that to the best of our knowledge, our technique is the first to:

(1) Statically and automatically construct, from an API source code, a model that accepts a superset of the legal API traces.
(2) Statically and automatically construct a model suitable for human inspection.

A more detailed discussion of related work follows.

Static Typestate Inference

Our technique is related to approaches that synthesize typestates [Strom and Yemini 1986; DeLine and Fahndrich 2001; Nanda et al. 2005] or interfaces [Alur et al. 2005; Giannakopoulou and Păsăreanu 2009; Henzinger et al. 2005] 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.

Aiming at verification imposes a safety requirement which tends to make abstractions overly restrictive in terms of the model behaviour. Permissiveness is possible only at the cost of assuming certain conditions over the artefacts being analysed, for instance the algorithms in [Giannakopoulou and Păsăreanu 2009; Henzinger et al. 2005] 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 Fig. 6 of [Alur et al. 2005]. This abstraction for the `ListItr` class 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 that omits most of the iterator behaviour and would be of little use for validation purposes.

In [Nanda et al. 2005] the authors present a technique to statically infer safe typestates in the presence of inter-object references. This approach is based on a mixture of predicate abstraction and abstract interpretation, and does not require the class internal state to be finite. However, like in the other approaches that we mentioned, the results obtained are aimed at creating test drivers and performing verification of client code. The result is accompanied with plenty of information regarding the boolean values obtained in the predicate abstraction process. The obtained amount of detail, while it helps to construct tests or guide verification processes, may hinder human-in-the-loop tasks such as visual inspection.

Approaches to perform modular verification of typestate usage (e.g., [Bierhoff and Aldrich 2008]) are based on annotating both the protocol and the client class with pre and postconditions (among other clauses). In general, the annotations for the protocol class can be manually generated since they are created once and used several times. On the other hand, there are thousands of different programs where a protocol is used and it is very time consuming to manually annotate all of those. In [Beckman and Nori

Technique	Input	Output	Construction	Purpose
[Alur et al. 2005]	API source code	Model that accepts subset of legal traces	Predicate abstraction, language learning	Verification of API client usage
[Nanda et al. 2005]	API source code	Model that accepts subset of legal traces	Predicate abstraction, abstract interpretation	Verification of API client usage
[Henzinger et al. 2005]	API source code (finite internal state)	Model that accepts all legal traces	Predicate abstraction via software model checking	Verification of API client usage
[Giannakopoulou and Păsăreanu 2009]	Finite LTS	Model that accepts all legal traces	Software model checking, language learning	Compositional verification
[Graf and Saïdi 1997]	Set of guarded-assignments, set of predicates	Model that accepts superset of legal traces	Predicate abstraction via assisted theorem proving	Verification of system properties
[Grieskamp et al. 2002]	Abstract state machine	Underapproximation of the "true FSM"	Symbolic execution	Construction of test-suite
[Liu et al. 2007]	API source code	Partition of concrete states according to the output of boolean observers	SMT solvers, testing	Construction of API test-suite
CONTRACTOR	API source code, requires clauses and invariant	Model that accepts superset of legal traces	Predicate abstraction using enabledness of operations	Human inspection
[Gabel and Su 2008]	API client traces	Model that accepts superset of observed traces	BDD-based mining algorithm	API specification recovery
[Dallmeier et al. 2006]	API client traces	Partition of concrete states according to observers output	Predicate abstraction over given traces	Construction of API test-suite
[Ghezzi et al. 2009]	API client traces	Model that accepts superset of given traces	Extrapolation via graph transformation rules	API specification recovery
[Lorenzoli et al. 2008]	API client traces	Model that accepts superset of given traces, preserving data dependencies	k-tail, data invariant inference	Construction of API test-suite
[Pradel and Gross 2009]	API client traces	Model that accepts superset of observed traces	States model methods, edges model precedence frequency between methods	API specification recovery
[Beschastnikh et al. 2011]	API client traces	Model that accepts superset of observed traces	Extrapolation via transitive closure of temporal invariants	Human inspection
[Demsky and Rinard 2009]	API client traces	Model that accepts superset of observed traces	Predicate abstraction using a set of built-in predicates	Human inspection

Table V: Related work summary

2011] the authors present a technique to automatically infer annotations for the client usages of the protocol.

Predicate Abstraction and Model Minimisation

Our work can be considered an instantiation of the predicate abstraction [Uribe 1999] framework. In this setting our work is related to techniques that construct abstract

state graphs from infinite state systems (e.g., [Lee and Yannakakis 1992; Graf and Saïdi 1997; Grieskamp et al. 2002]). 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 [Graf and Saïdi 1997] to model the enabledness conditions of actions, the output would be too large for manual inspection (see [de Caso et al. 2010] for further discussion). Notably, the setting in [Grieskamp et al. 2002] admits producing the same abstraction as ours for testing purposes but the approach is to under-approximate it by finitely bounding the artefact under analysis.

Testing-related Approaches

A level of abstraction somewhat related to that of enabledness has been used in [Liu et al. 2007]. The authors quotient the state space of a class based on 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. These models are then fed to an algorithm that attempts to create a test suite that covers all of the states. Our work differs in two significant ways: *(i)* their approach constructs the set of states using (a subset of the) class observers while we rely on (all of the) class methods that change its state; and *(ii)* we do not require the presence of a representative set of boolean observers in order to produce an abstraction. The abstraction produced in [Liu et al. 2007] 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.

In [Grieskamp et al. 2008] the state space of a model given by a set of precondition-guarded actions is explored. They do not intend to construct a complete finite abstraction out of it, but to explore it in order to generate test cases.

Model Mining

Our approach relates to the mining of temporal specifications (e.g., [Dallmeier et al. 2006; Ghezzi et al. 2009; Gabel and Su 2008; Lorenzoli et al. 2008; Dallmeier et al. 2010; Beschastnikh et al. 2011; Pradel and Gross 2009]), 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 a 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 over-approximations 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.

The main difference with [Gabel and Su 2008] is that the resulting automata are built from the client's actual usage of a set of operations rather than from the constraints of usage provided by requires clauses.

Tools such as ADABU [Dallmeier et al. 2006] produce finite state machines whose states are determined by a fixed level of abstraction ranging over the return values of the inspectors in a class. For instance, integers are abstracted according to its sign, therefore this technique is not suitable for differencing two significant concrete program states distinguished by a different positive integer. Our approach depends on the preconditions in order to create the set of states; if preconditions mention specific integer values then BLAST is going to consider them for us.

In [Ghezzi et al. 2009], a way to generalise component behaviour using samples taken during a systematic bounded execution is presented. In a first step a deterministic finite state machine is built using the sampled behaviour. This is then generalised using graph transformation rules and invariant detection tools. If an implementation

were to be sampled using this technique then we would end up having a set of graph rules tightly correlated to the original artefact. That is, the technique would traverse the inverse path we define in our work.

A similar approach can be found in [Lorenzoli et al. 2008], a technique in which behavioural models that preserve data and control dependencies are mined out of execution traces. In a first step, sets of traces that share the same actions are identified and their parameters are abstracted away by applying DAIKON. This produces a tree-like representation in which then states are joined if they share a common k-future. These techniques unsoundly generalise observed behaviour by applying invariant detecting tools. Unlike our approach, the amount and quality of behaviour space synthesized depend on the traces used as input. On the other hand, there is no clear indication that yielded abstractions would be coarse enough for validation. The models we produce can be seen as the k-tail abstraction [Lorenzoli et al. 2008] (with $k = 1$) of the infinite set of traces for a given program.

Finally, [Pradel and Gross 2009] introduces a similar mining approach, but avoids the approximation introduced by learning algorithms. Each state in the model they produce is mapped to a single method. A transition between two states (methods) is added whenever one method is invoked after the other in an observed trace. Weights are used to distinguish the most frequently observed method interactions.

Models Aimed at Human Inspection

As we previously stated, most of the models used in the typestate and interface synthesis literature are used to feed engineering tasks such as verification and test-case generation. These approaches build a model suitable for verification at the cost of either: *(i)* aiming at verification of client code; or *(ii)* targeting a particular property φ.

With respect to *(i)*, even when it is an interesting and challenging problem, we are currently not interested in checking client usage of an API. We are focused in helping the developer determine if the API implementation provides (and only provides) the intended services. Determining this is prior to deciding if a client does proper use of those services.

Regarding *(ii)*, while it is sometimes taken for granted that a φ to be checked against the API implementation exists, it is usually not a trivial problem getting such φ. In some cases it is hard to come up with the given property in the first place. In many cases the desired property is informally specified. How do we know that φ is a correct formalization of the intended property? Even when having a correct formalized φ, how do we know if it is enough, on its own, to guarantee that the API implementation provides the intended services to its clients? Sometimes it suffices to verify 2 or 3 properties, but how do we know if a set of properties Φ is enough?

If the developer has a property ϕ in mind, then EPAs may not have the best level of abstraction to determine if such property holds. On the other hand, if the developer does not have any property in mind, but instead wants to get a quick overview of the behaviour space of the API implementation, EPAs can provide a good starting point.

There are other approaches that, similarly to ours, aim at constructing models for validation. For instance, the approach followed in [Beschastnikh et al. 2011] uses logging mechanisms already in place and regular expressions to obtain behaviour models without too much user intervention. The logs are mined looking for invariants encoding simple temporal restrictions among operations. Then, models are produced such that they satisfy every invariant found in the previous step. The results obtained in this case, similarly to ours, have been successfully used to guide human validation processes such as program understanding or bug confirmation. However, the tool presented at [Beschastnikh et al. 2011] requires a logging mechanism in place, something which is not generally available in an early development stage on which we envision

CONTRACTOR being applied. We therefore think this approach is complementary to ours.

Another example of synthesised models being used for human inspection is introduced in [Demsky and Rinard 2009]. Authors present a technique to dynamically construct role transition diagrams (among other models), which have a resemblance to typestates. These models are used, together with a powerful graphical user interface, to support program understanding tasks.

In [de Caso et al. 2010] the authors 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 [Leavens et al. 2007]. 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 4.2 together with the over and under-approximated requires clauses, which allowed us to use a software model checker for the EPA construction.

In [Zoppi et al. 2011] we also explore the possibility to construct EPAs using verification-based approaches. In particular, we implemented an experimental tool that uses CODE CONTRACTS [Andersen et al. 2009] to build EPAs out of C# programs. We also present an experimental technique to verify proper API client usage using the inferred EPAs.

The DAIKON tool [Ernst et al. 2007] could have been used to obtain a program's contract and use the technique we presented in [de Caso et al. 2010] to generate its enabledness preserving abstraction. However, postconditions are complex logic formulas in general and accurate inference is unfeasible in practice. The technique presented in this paper does not require postconditions to be explicit, therefore we could benefit from DAIKON's potential to infer good preconditions, while not having to endure the problem of obtaining postconditions. Nevertheless, we could still use Daikon to automatically obtain requires clauses and invariants. We would still have to take into account that the assertions that DAIKON outputs are true for the runs that it used to create them, yet not necessarily true for all the possible runs.

Alternatively, we could use the function summaries that tools such as BLAST internally construct. These summaries have a strong resemblance of pre/postcondition contracts, therefore enabling the use of the methodology presented in [de Caso et al. 2010].

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.

We plan to conduct empirical studies in order to evaluate how useful are EPAs for various software engineering activities such as bug finding, program understanding or test-case generation. Other possibilities involve studying how these activities are impacted when using incomplete EPAs (i.e., EPAs that fail to capture some fragment of the legal behaviour).

We are currently studying the application of EPAs in model-based testing. In particular, we envision that EPAs can be used to derive coverage criteria. EPAs could also drive the test-case generation in order to reach all abstract states (or transitions), therefore avoiding to miss corner cases.

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 [Leavens et al. 2007] 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.

REFERENCES

ALUR, R., ČERNỲ, P., MADHUSUDAN, P., AND NAM, W. 2005. Synthesis of interface specifications for Java classes. In *POPL '05*. 98–109.

ANDERSEN, M., BARNETT, M., FAHNDRICH, M., GRUNKEMEYER, B., KING, K., LOGOZZO, F., PATEL, V., AND ZUNIGA, D. 2009. Code Contracts.

BARRETT, C. AND BEREZIN, S. 2004. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16 th International Conference on Computer Aided Verification (CAV '04)*. 515–518.

BECKMAN, N. AND NORI, A. 2011. Probabilistic, modular and scalable inference of typestate specifications. *PLDI*.

BECKMAN, N. E., KIM, D., , AND ALDRICH, J. 2011. An empirical study of object protocols in the wild. In *ECOOP 2011*.

BESCHASTNIKH, I., BRUN, Y., SLOAN, S., AND ERNST, M. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *FSE 2011*.

BEYER, D., HENZINGER, T., JHALA, R., AND MAJUMDAR, R. 2007. The software model checker Blast. *STTT 9*, 505–525.

BIERHOFF, K. AND ALDRICH, J. 2008. Plural: checking protocol compliance under aliasing. In *ICSE*. ACM, 971–972.

COK, D. AND KINIRY, J. 2005. ESC/Java2: Uniting ESC/Java and JML. *Lecture Notes in Computer Science*, 108–128.

DALLMEIER, V., KNOPP, N., MALLON, C., HACK, S., AND ZELLER, A. 2010. Generating test cases for specification mining. In *ISSTA 2010*.

DALLMEIER, V., LINDIG, C., WASYLKOWSKI, A., AND ZELLER, A. 2006. Mining object behavior with ADABU. In *Workshop on Dynamic systems analysis '06*.

DE CASO, G., BRABERMAN, V., GARBERVETSKY, D., AND UCHITEL, S. 2010. Automated abstractions for contract validation. *TSE*.

DELINE, R. AND FAHNDRICH, M. 2001. Enforcing high-level protocols in low-level software. In *PLDI '01*. 59–69.

DEMSKY, B. AND RINARD, M. 2009. Automatic extraction of heap reference properties in object-oriented programs. *IEEE Transactions on Software Engineering 35*, 305–324.

DIJKSTRA, E. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18,* 8, 453–457.

ERNST, M., PERKINS, J., GUO, P., MCCAMANT, S., PACHECO, C., TSCHANTZ, M., AND XIAO, C. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69*, 35–45.

GABEL, M. AND SU, Z. 2008. Symbolic mining of temporal specifications. In *ICSE '08*. 51–60.

GHEZZI, C., MOCCI, A., AND MONGA, M. 2009. Synthesizing intensional behavior models by graph transformation. In *ICSE '09*. 430–440.

GIANNAKOPOULOU, D. AND PĂSĂREANU, C. 2009. Interface generation and compositional verification in JavaPathfinder. In *FASE '09*. 94–108.

GRAF, S. AND SAÏDI, H. 1997. Construction of abstract state graphs with PVS. In *CAV '97*. 72–83.

GRIESKAMP, W., GUREVICH, Y., SCHULTE, W., AND VEANES, M. 2002. Generating finite state machines from abstract state machines. In *ISSTA '02*. 112–122.

GRIESKAMP, W., KICILLOF, N., MACDONALD, D., NANDAN, A., STOBIE, K., AND WURDEN, F. 2008. Model-based quality assurance of windows protocol documentation. In *ICST '08*. 502–506.

GRIESKAMP, W., KICILLOF, N., STOBIE, K., AND BRABERMAN, V. 2011. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability 21,* 1, 55–71.

HENZINGER, T., JHALA, R., AND MAJUMDAR, R. 2005. Permissive interfaces. In *ESEC/FSE '05*. 31–40.

HODGES, W. 1997. *A shorter model theory*. Cambridge University Press, Cambridge New York.

KHURSHID, S., PĂSĂREANU, C., AND VISSER, W. 2003. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for the Construction and Analysis of Systems*, 553–568.

KLENSIN, J., FREED, N., ROSE, M., STEFFERUD, E., AND CROCKER, D. 1995. Smtp service extensions. Tech. rep., RFC 2846, November.

LEAVENS, G., LEINO, K., AND MÜLLER, P. 2007. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing 19,* 2, 159–189.

LEE, D. AND YANNAKAKIS, M. 1992. Online minimization of transition systems (extended abstract). In *STOC '92*. 264–274.

LIU, L., MEYER, B., AND SCHOELLER, B. 2007. Using contracts and boolean queries to improve the quality of automatic test generation. In *TAP '07*. 114–130.

LORENZOLI, D., MARIANI, L., AND PEZZÈ, M. 2008. Automatic generation of software behavioral models. In *ICSE '08*. 501–510.

NANDA, M., GROTHOFF, C., AND CHANDRA, S. 2005. Deriving object typestates in the presence of inter-object references. *ACM SIGPLAN Notices 40,* 10, 77–96.

PACHECO, C. AND ERNST, M. 2007. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 815–816.

PRADEL, M. AND GROSS, T. R. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *ASE 2009*. IEEE, 371–382.

STROM, R. AND YEMINI, S. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE 12,* 1, 157–171.

URIBE, T. 1999. *Abstraction-based Deductive-algorithmic Verification of Reactive Systems*. Stanford University, Dept. of Computer Science.

ZOPPI, E., BRABERMAN, V., DE CASO, G., GARBERVETSKY, D., AND UCHITEL, S. 2011. Contractor.net: inferring typestate properties to enrich code contracts. In *Proceeding of the 1st workshop on Developing tools as plug-ins*. TOPI '11. ACM, New York, NY, USA, 44–47.