Abstractions for Validation in Action

Guido de Caso[1], Victor Braberman[1], Diego Garbervetsky[1],
and Sebastian Uchitel[1,2]

[1] Departamento de Computación, FCEyN, Universidad de Buenos Aires,
Buenos Aires, Argentina
{gdecaso,vbraber,diegog,suchitel}@dc.uba.ar
[2] Department of Computing, Imperial College,
London, UK
s.uchitel@doc.ic.ac.uk

Abstract. Many software engineering artefacts, such as source code or specifications, define a set of operations and impose restrictions to the ordering on which they have to be invoked. Enabledness Preserving Abstractions (EPAs) are concise representations of the behaviour space for such artefacts. In this paper, we exemplify how EPAs might be used for validation of software engineering artefacts by showing the use of EPAs to support some programming tasks on a simple C# class.

Keywords: Behaviour validation, enabledness-preserving abstractions.

1 Introduction

Verification and validation are artefact evaluation activities that are carried out by software engineers in multiple stages of software development projects. They come in many different guises: The artefacts under evaluation may be descriptions related to the problem domain (e.g. requirements) or the solution space (e.g. design) including the actual code. Furthermore, they can be written in languages with different degrees of formality (e.g. from mathematics to natural language). In addition, the evaluation itself can vary in terms of formality (e.g. from axiomatic proof, through structured argumentation, to human inspection) and exhaustiveness (e.g. from exhaustive search, through simulation, to selective scenario evaluation). All these characteristics lead to conclusions with very different degrees of certainty.

Verification and validation are related activities both of which aim to increase confidence regarding the quality of the software under construction. However, they are of very different nature.

1.1 Verification

Verification aims at determining whether an artefact satisfies specific properties [25]. For instance, if software requirements entail system goals, if the architecture satisfies its reliability requirements, if the code is structured according to the static design, or the execution of a method never raises an array index out of bounds exception.

Verification is particularly prone to automated, rigorous and even sometimes exhaustive analysis. If both the artefact under evaluation and the properties are given in appropriate formal languages, it is plausible to apply a battery of tools such as model checkers [6], theorem provers [32], simulators [27] or symbolic executers [34]. There are, of course, both theoretical (indecidability results, e.g., [16]) and practical (e.g., state explosion [37]) limitations. However, automated verification techniques are tractable and have shown to be useful, specially when applying some restrictions on the artefact, the property, and/or the degree of certainty. Most notably, software testing, when the intended test results are provided (i.e., an oracle), is a widespread verification technique in industry.

1.2 Validation

Validation is, arguably, a much more complex task than verification as it aims to determine the degree to which an artefact is an accurate representation of the real world. At the requirements level, a typical example used to distinguish validation from verification is that validation evaluates if the requirements meet stakeholders needs, while verification is applied to check that the design and/or implementation has been built according to the requirements. In other words, validation ensures that you built the right thing while verification ensures that you built it right. Validation is indeed relevant in many software engineering settings. For instance, determining if an architectural description conforms to an architect's intent, if the deployment model is consistent with the actual hardware available at the client site, if assumptions on network traffic are reasonable, etc.

Validation, in industrial practice, is also a substitute for verification. The lack of explicit (formal or informal) intended property descriptions impedes verification and the only possibility is to validate if artefacts conform to the characteristics intended by the engineer. In other words, a comparison between the artefact and some mental model of what the artefact should comply to. Walkthroughs, inspections and reviews are common techniques that support validation.

When the artefact under validation is written in a formal language (be it code, or a well founded specification), a common strategy for validation is to apply an automated, semantics preserving, manipulation. The idea behind this strategy is that showing engineers alternative views of the artefact may exhibit elements that stand out as contradicting to what is expected by the engineers. Some examples of this strategy are the application of rewrite rules in specifications, minimisation of state machines, slicing techniques for code, or executions and simulations. Within the latter strategy, testing without oracles is a noteworthy technique.

Another common strategy for validation is to turn the validation problem into a verification one. More concretely, to produce a specification against which the artefact can be verified. The idea is that if the specification is simpler than the artefact, validation of the former is likely to be simpler and less error prone. This is an effective strategy that is commonly used in practice. For instance, sanity checks are used to filter out bugs in complex models (such as nonzenoness in real time system models [2], as well as internal consistency and satisfiability in requirements specifications [23]). However, this strategy has its downsides too. Since an alternative specification is required, we need to be sure that it has been validated appropriately. In other words, turning a validation problem into a verification problem creates a new (possibly simpler but of reduced scope) validation task, so eventually human intervention is required.

1.3 Abstraction

Abstraction is the act of withdrawing or removing something, the process of leaving out of consideration one or more properties of a complex artefact so as to attend to others. It is also used to refer to the simpler artefact that results from this process. The abstraction captures the original artefact's core or essence relative to a specific aspect of interest. Abstraction is central to computing [26], particularly to software engineering, and has been extensively applied to support verification and validation.

Abstraction reduces the complexity of the artefact under evaluation and consequently can reduce the cost and augment the effectiveness of verification and validation activities. However, abstraction comes at a price. Building abstractions can be costly, but perhaps more importantly, the loss of detail in the abstract artefact can impact the degrees of certainty of the evaluation outcome. Given a particular verification or validation task, analysing a carefully chosen abstraction will yield conservative (yet sound) results. On the other hand, an incorrect choice might lead to invalid conclusions. For instance, let's consider a language with automatic memory management. A garbage collector (GC) is in charge of reclaiming unreferenced objects. In order to make a decision whether an object o can be collected, the GC must ensure that no other object or variable points to o. If the GC makes the decision based on an abstraction that only considers elements in the program stack, it may collect objects that are still reachable from the heap.

Hence, given a validation or verification task, it is crucial to work with an appropriate abstraction. That is, carefully selecting which aspects to leave out of consideration and what mechanisms to use for representing the artefact's features relevant to the task at hand.

1.4 Abstractions for API Implementation Behaviour

We now set the scene for the rest of this paper. We first narrow the discussion by stating our interest in the behaviour of stateful API implementations: A collection of methods or procedures accessing a shared data structure. Such artefacts are commonplace and pose a number of challenges to verification and validation. It is insufficient to evaluate each method in isolation. We also need to consider their interaction via the shared data structure and the emergent behaviour of the combinations of method calls.

State machines or *behaviour models* constitute a natural abstraction to validate or verify an API implementation behaviour. Each abstract state in these models represents one or more possible concrete states of the shared data structure. Transitions in the abstraction represent changes of state related to the execution of one or more lines of code (including, for instance a whole method invocation).

Abstractions for Verification. There has been a significant amount of work in the use of abstractions for *verification*. Given an artefact a and a formalised property φ to be verified, the aim is to automatically come up with simplified versions $\hat{a}$ and $\hat{\varphi}$ of the artefact and property, respectively. Hopefully, verifying whether $\hat{a}$ satisfies $\hat{\varphi}$ will be more tractable, while still providing information about the initial verification problem regarding a and φ.

Applying abstraction to obtain a tractable behaviour model of the original artefact typically involves paying the cost of the omitted detail in terms of loss of precision. Abstracted behaviour models may be overapproximations (when $\hat{a}$ accepts all behaviour of a, but possibly more) or underapproximations (when $\hat{a}$ rejects all behaviour not in a, but possibly more) of the original artefact. Furthermore, some abstractions may neither be over nor underapproximations.

Given an API implementation, an overapproximation of its behaviour describes all legal invocation sequence that API clients can perform on the API. However, an overapproximation may include sequences which, if performed by clients, would result in illegal invocation chains. On the other hand, an underapproximation of the API implementation's behaviour forbids every illegal invocation sequence, which is why they are called *safe* from a client perspective. Underapproximated models may go too far and forbid behaviour which was permitted in the original artefact. For this reason, overaproximated models are sometimes referred to as *permissive*.

One common approach when applying abstraction for verification of API behaviour is to synthesise typestates [35,13,31,4] or interfaces [1,20,24]. The aim is to statically obtain finite state machine representing a *safe* model from a client perspective, using techniques such as automata learning [1,20,24] or abstract interpretation [31].

The safety requirement associated to these kind of models tends to make abstractions overly restrictive in terms of the model behaviour, sometimes leading to trivial abstractions (e.g. models in which very few or even no invocation sequences are allowed). In some cases, permissiveness is possible at the cost of assuming certain conditions over the artefacts. For instance, the algorithms in [20,24] guarantee permissiveness only when the library's internal state is finite.

Once inferred, safe typestates for an API can be used to effectively verify the absence of illegal invocations from clients (e.g., [7]). The cost of non-permissive typestates in this setting is that false-positives (client invocation sequences that are in fact legal) may be reported.

Another way of obtaining abstract behaviour models is by using predicate abstraction [36]. The idea is to define a set of predicates P and group concrete states according to the validity of those predicates. Concretely, each abstract state represents a set of concrete states that gives the same valuation to all the predicates in P.

There are techniques that use this approach to construct abstract state graphs from infinite state systems (e.g., [28,21,29,22]). For instance [21] builds an abstract state graph out of a guarded transition system and a set of input predicates. Concrete states are abstracted by using a lattice of monomials of abstract boolean variables representing the truth values of the input predicates.

For testing purposes, [29] proposes the use of user-provided parameterless boolean observers to quotient the state space of a class. 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 (which may not be fulfilled due to the overapproximated nature of the abstraction) . These models are then fed to an algorithm that attempts to create a test suite that covers all of the states.

Another interesting approach is the mining of behaviour models out of execution traces (e.g., [11,19,18,30,12,5,33]). These techniques aim at inferring a specification which is used for test case generation or verification.

Mining techniques have a dynamic flavour, and thus heavily depend on the quality of the traces used as input. The inferred models tend to be underapproximations of the behaviour of the artefact under analysis, since some behaviour may not appear in the collected traces. However, in some cases, these approaches may also over-approximate due to the application of generalisation strategies.

For instance, [18] produces an automata by collecting information from the client's actual usage of a set of operations (underapproximation). Adabu [11] produces finite state machines whose states are determined by a fixed level of abstraction ranging over the return values of the inspectors in a class (e.g., integers are abstracted according to its sign), leading to both under and overappproximation of the concrete state. Another approaches [19,30], use invariant detection tools such as Daikon [15] in order to generalise the set of traces and obtain more conservative models.

Abstractions for Validation. We are interested is studying the use of abstraction in the context of validation rather than verification. Since validation requires human intervention, the size and complexity of the models obtained are a key aspect at the moment of choosing the abstraction.

As we previously stated, most of the models used in the typestate and interface synthesis literature are used to feed machine-driven tasks such as automated verification and test-case generation. There are a few exceptions, though.

For instance, the approach followed in [5] uses logging mechanisms already in place and regular expressions to obtain behaviour models almost without user intervention. The logs are mined looking for invariants encoding simple temporal restrictions among operations. Then the authors build a behaviour model that satisfies every invariant found in the previous step. These models have been successfully used to guide human validation processes such as program understanding or bug confirmation.

Another example of synthesised models being used for human inspection is introduced in [14]. 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.

Even though these examples show the use of underapproximations for the validation of artefacts, we believe that in general overapproximations are better suited for validation since they are capable of exhibiting all the potential behaviour of the artefact.

In this paper we will study a particular abstraction level that we denominate *enabledness-based abstractions* [8] that focus the attention on the enabledness of a set of actions or method within a API. We have successfully evaluated the potential of these abstractions both for contract specifications validation [8] as well as for validating code implementing APIs [9].

This rest of this paper is structured as follows: Section 2 introduces enabledness-preserving abstractions (EPAs); Section 3 presents how the use of EPAs can aid software developers in various activities such as program understanding or the implementation of new features; Section 4 provides a series of strategies and checklists to work with EPA-guided software validation; finally, Section 5 concludes our presentation.

2 Background

Enabledness-preserving abstractions (EPAs) are state machines that describe behaviour of API implementations by introducing abstraction in two different ways. Firstly, states reached by the implementation while executing API operations are ignored; focus is on states of the implementation before and after operations are executed. Secondly, the actual values of the data structures of the implementation are abstracted; focus is what operations those values allow or disallow. Hence, states of an EPA represent sets of concrete implementation states which allow the same set of operations. Finally, parameters and return values of operations are also ignored; focus is on whether there exist values for parameters of an operation such that the execution of the operation will make the implementation transition from one abstract state to another.

We briefly define EPAs more formally utilising object oriented terminology. For a more detailed presentation see [9]. A *class* C can be seen as a structure $\langle M,\ F,\ R,\ inv,\ init \rangle$, where $M = \{m_1, \ldots, m_k\}$ is a finite set of *public method labels*, F is an M-indexed set of *method implementations*, R is an M-indexed set of *requires clauses*, *inv* is the *class invariant*, and *init* denotes the *initial conditions* given by the constructors. Given a class C and two instances c_1, c_2, we say that c_1 and c_2 are *enabledness equivalent* (noted $c_1 \equiv_e c_2$) iff for every $m \in M$: c_1 satisfies R_m iff c_2 satisfies R_m (i.e. if the two instances satisfy the same set of requires clauses).

The set of class instances, when quotiented by $\equiv_e$, results in a set of abstract states, such that each one is mapped to a (distinct) group of enabled methods. Each abstract state groups all the instances that share the same set of enabled methods, and can be characterized by a *state predicate*. Formally, the predicate for an abstract state given by a set of methods $\mathsf{ms} \subseteq M$ is a function $\mathrm{pred}_{\mathsf{ms}}$ that takes an instance of C and returns a boolean. It is formally defined as:

$$\mathrm{pred}_{\mathsf{ms}}(c) \overset{\mathrm{def}}{\Leftrightarrow} \mathit{inv}(c) \ \wedge \bigwedge_{m \in \mathsf{ms}} c \text{ safisties } R_m \ \wedge \bigwedge_{m \notin \mathsf{ms}} c \text{ does not satisfy } R_m$$

An abstract transition labeled with an action m between two abstract states exists if and only if a class instance in the target abstract state can be reached by executing m from a class instance in the source abstract state. Formally, let ms_1 and ms_2 be method sets (that is, abstract states) and $m \in \mathsf{ms}_1$ an action. An m-labeled transition is added from ms_1 to ms_2 if there is a class instance c_1 in ms_1 that can execute m and evolve into a class instance c_2 in ms_2.

EPAs capture a superset of the concrete class' behaviour. In practice, this level of abstraction provides a good compromise: it is abstract enough to keep the EPAs concise, and it is precise enough so that it is still a valuable information source for humans to inspect.

2.1 EPA Construction

Enabledness-preserving abstractions can be statically and automatically built. While the details of the construction process are out of scope in this presentation, we present some brief notes and pointers to other articles.

The reader may notice that a class with k public methods has potentially 2^k reachable abstract states. A naïve construction algorithm would compute all the 2^k states and its transitions, only to later restrict the result to the reachable fragment. On the other hand, more sophisticated approaches such as parallelized BFS exploration strategies (e.g. [9]) can drastically reduce construction times.

We implemented a prototype tool[1] called Contractor which implements various EPA construction algorithms, together with several optimisations. In its current version, Contractor supports the construction of EPAs directly from pre/postcondition contracts [8], C code [9] and .NET code [38].

In the rest of this paper, we focus on constructing EPAs out of C# code, one of the most popular .NET languages. We leverage existing .NET infrastructure to let the programmer identify the requires clauses and the class invariant. More specifically, we use the Code Contracts [3] library calls `Contract.Requires(...)` and `Contract.Invariant(...)`.

3 Developing with EPAs

In this section we describe some simple (and fictional) software development tasks supported by using enabledness-preserving abstractions as a visual aid. Although applied to a small class, we aim to exemplify how EPAs can aid in various tasks such as understanding someone else's source code, extending it

[1] Publicly available at `http://lafhis.dc.uba.ar/contractor`

```
public class Stack {
  private int[] elems;
  private int next;

  public Stack(int m) {
    next = 0;
    elems = new int[m];
  }

    public int top() {
      if (isEmpty())
        return -1;
      else
        return elems[next - 1];
  }

    public bool isEmpty() {
      return next == 0;
  }

    public int maxSize() {
      return elems.Length;
  }
  public bool isFull() {
    return next == elems.Length;
  }

  public void push(int k) {
    elems[next] = k;
    next++;
  }

  public void pop() {
    if (isEmpty())
      return;
    next--;
  }

  public void reset() {
    next = 0;
  }
}
```

Fig. 1. Train door controller

with a new feature, testing the new functionality, debugging the problems that may arise and fixing them. Interested readers are directed to [8,9] for a report on the application of EPAs on various real-life industrial scale software artefacts.

3.1 Understanding Code with EPAs

Tom, our fictional programmer hero, has just taken a job at DataXtructures Inc., a fictional software developing company specialized in implementing efficient data containers.

His first assignment is to extend the functionality of a `Stack` class with some new operations. The original code for the class that Tom has to extend is depicted in Figure 1.

After a brief code inspection, Tom soon realizes that this simple data structure was implemented using an underlying array. This imposes an upper bound on the number of elements that can be stored. He also discovers that the are a number of observer operations such as `top()`, `isEmpty()` or `maxSize()`. There are also mutator operations such as `push(int k)`, `pop()` and `reset()`.

While each method is at most a couple of lines long, understanding how these operations can be interleaved poses a much bigger challenge. For instance, it is reasonable to expect `pop()` to be an illegal operation on a freshly constructed `Stack` instance. Similarly, it would be awkward if that same operation was illegal after successfully pushing an element on any legal stack.

This restrictions and many others constitute a set of “common sense” behaviour that any programmer would expect from a stack. We refer to this set of informal (and often implicit) requirements as *mental model.*

The Stack class has been around in the company for a while, so Tom is quite confident that it behaves properly (this is, according to his mental model). Still, he wants to double-check this so he writes a couple of small programs such as the following.

```
public void Main(string[] args) {
  Stack s = new Stack(10);
  s.push(27); s.push(19);
  System.Console.WriteLine(s.top());
  s.pop();
  System.Console.WriteLine(s.top());
}
```

Tom manually compares the output with his mental model, effectively confirming that 19 is printed first, followed by 27.

At this point Tom has a small positive piece of evidence that the Stack class behaviour is aligned with his mental model. The problem for Tom is how to confirm his hunch, how to know that he tested enough. In other words, he might wonder *"how do I know that this is indeed a correct stack implementation? When do I stop testing/inspecting?"*.

Fortunately, a colleague told Tom a little bit about enabledness-preserving abstractions and Tom is willing to give them a try. By statically constructing an EPA from the Stack source code, Tom will be able to compare his mental model with the graphical representation of the abstraction and decide if there are any inconsistencies between his understanding and the implementation.

As we mentioned before, in order to build an EPA for a class C we need to define its components: a set of methods names M, their implementations F, their requires clauses R, the class invariant inv and the class initial condition $init$.

The relevant operations that compose M are the mutators `push(int k)`, `pop()` and `reset()`. The constructor `Stack()` is used to define the initial condition $init$, and the rest of the operations are merely observers, which do not affect the internal state of the stack and which are excluded in the rest of the section to simplify presentation.

The implementation set F for the selected group of operations is given by the appropriate fragments of code in Figure 1. For instance, lines 28–31 define the implementation for the `push` operation.

Regarding the requires clauses, for each operation m Tom needs to identify (or explicitly add, if necessary) the adequate fragment of code that performs the parameter and/or fields validation. This code fragment is then converted to a boolean expression `b` that guarantees a safe invocation of m. Finally, Tom needs to prepend the code for m with `Contract.Requires(b)`.

For instance, lines 34–35 of Figure 1 check that the stack is not empty while attempting to invoke the `pop` operation. Tom then rewrites this check as a `Contract.Requires(...)` invocation, which results in the following.

```
public void pop() {
  Contract.Requires(!isEmpty());
  next--;
}
```

When considering the `push` operation there are no explicit fragments of code that perform validation. However, this omission is actually a bug. The problem is that the stack is based on a fixed-size array and pushing infinitely would eventually produce an exception when the index goes beyond the limits of the array.

In order to fix this, Tom adds a `Contract.Requires(...)` check to the method as follows.

```
public void push(int k) {
  Contract.Requires(!isFull());
  elems[next] = k;
  next++;
}
```

Finally, since it is safe to invoke the `reset` operation on any stack, there is no need to rewrite it.

Having fixed the requires clauses for the relevant operations, Tom still needs to define a class invariant *inv* and an initial condition *init*. As a first step, Tom decides to leave the invariant as `true`. In other words, any possible value assignment of the `Stack` class fields represents a valid stack.

Regarding the initial condition, it is straightforward to discover from the source code of the `Stack` constructor that initially `next == 0` and `elems` is a fresh integer array. The size of `elems` depends of the actual value of the constructor parameter `m`, but Tom realizes that empty arrays make no sense. He therefore decides to always leave room for at least 2 elements and adds a `Contract.Requires(...)` check to the constructor as follows.

```
public Stack(int m) {
  Contract.Requires(m > 1);
  next = 0;
  elems = new int[m];
}
```

Having defined the action names, their implementations and requires clauses, as well as the system invariant and initial condition, Tom is now ready to run Contractor and obtain an EPA for the `Stack` class. Notice that mining the requires clauses might seem an unnecessary burden (which could by lessened to some extent with dynamic inference and static code analysis techniques), but it actually led Tom to the discovery of a bug in the `push` operation.

Figure 2 presents the `Stack` EPA. As we mentioned, each abstract state is described by a set of operations and groups all concrete stacks that enable those operations only.

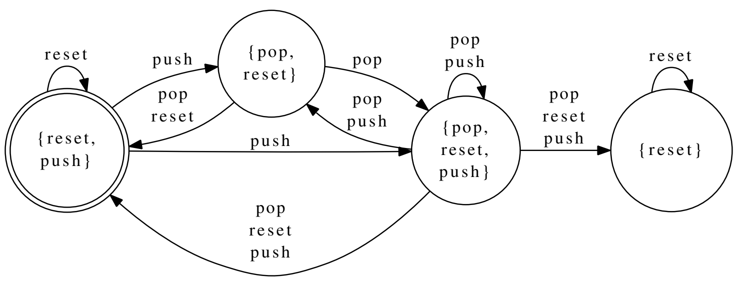

Fig. 2. Initial EPA for the `Stack` class

The initial abstract state is marked with a double circle and enables both the `reset` and `push` operations. This seems reasonable to Tom since `reset` is always enabled and a freshly constructed Stack always has room for at least one element, making `push` enabled as well. Furthermore, popping elements from a freshly constructed empty stack is illegal, which is also aligned with Tom's mental model.

Pushing from the initial state is a non-deterministic action. It can take the `Stack` instance to two possible destinations:

1. The {`pop`, `reset`} state, on which pushing is no longer enabled. This abstract states groups the full stacks.
2. The {`pop`, `reset`, `push`} state, on which pushing is still enabled. This is the abstract state that groups stacks that are 'half-full'.

Tom notices however, that since the minimum size for a `Stack` is 2, it is not possible to fill-up a freshly constructed instance after pushing a single element. The transition from the initial state to the full state is not feasible. Furthermore, while the initial abstract state and its outgoing transitions look rather good, there is a sink state {`reset`} on the right which seems suspicious. The presence of this abstract state indicates that eventually a `Stack` instance may be rendered virtually unusable, since both pushing and popping would never be enabled for the lifetime of that instance. Carefully reading the requires clauses for `push` and `pop`, Tom discovers that this could only happen if the `Stack` instance was full and empty at the same time. But this is not possible, so something is wrong.

These observations are not exhaustive, but even after a short inspection of the EPA Tom has discovered that stacks may present strange behaviour. However he feels pretty confident that it is impossible for a fresh `Stack` instance to get full after a single `push` operation. And it is as impossible for any `Stack` instance to be both empty and full at the same time.

Tom then remembers that he used an empty system invariant, and that might have affected the resulting EPA. He creates a refined invariant and adds it to the class as a new method as follows.

```
[ContractInvariantMethod]
private void Invariant()
{
  Contract.Invariant(0 <= next);
  Contract.Invariant(next <= elems.Length);
  Contract.Invariant(elems.Length > 1);
}
```

The ContractInvariantMethod attribute in line 1 is a Code Contracts keyword that identifies the method as the class invariant. Lines 4 and 5 indicate the bounds for the variable that points to the next free element in the array (if any). Notice that since popping is disabled on empty stacks, `next` will never be negative; and since pushing is not allowed on full stacks, then `next` can not be larger than the size of the `elems` array. Following the decision made when redefining the constructor, the last line of the invariant states that there is always room for at least two elements.

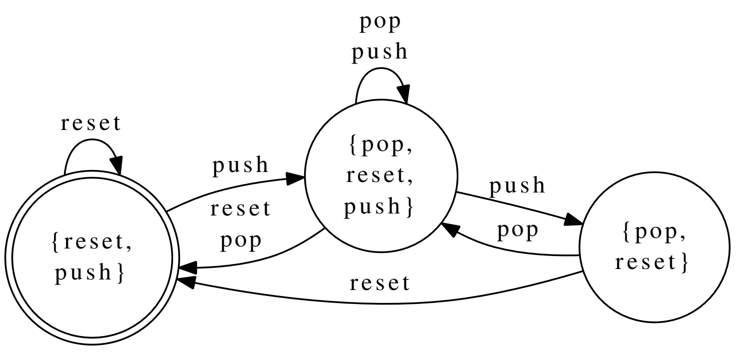

Fig. 3. EPA for the `Stack` class with proper invariant

Figure 3 presents the EPA that Tom gets when feeding Contractor with the newly defined invariant. Tom realizes that this EPA is very well aligned with his mental model. The suspicious sink state is now gone, and so is the `push` transition that connected empty stacks directly with full stacks. The remaining states and transitions are consistent with Tom's idea of how a `Stack` instance should behave. For instance, the `reset` operation always takes instances back to the initial state, pushing eventually leads to the full state and popping eventually takes instances back to the initial state.

However, not everything in Tom's mental model is covered in this EPA. For instance, the fact that the `Stack` shows actual LIFO (last in, first out) behaviour is not part of the abstraction. Likewise, the overapproximated nature of the EPA, implies it allows paths which are clearly illegal such as `push`⇝`pop`⇝`pop`. While EPAs convey a concise representation of a class state-space, the negative side-effect is that some aspects of the behaviour get lost in the way. More precise abstractions could capture some of these aspects that EPAs miss, but they would do so at the cost of larger representations that could get in the way when manually inspecting the abstraction.

3.2 Implementing and Debugging with EPAs

Now that Tom is confident with the Stack implementation being correct, he can proceed with his first assignment. Different projects inside DataXtructures Inc. make use of stacks, and some of them require the elements to be returned in an ordered fashion. In other words, Tom has to add a popMax() operation that returns the maximum integer on a Stack instance and removes it, leaving the rest of the elements untouched, preserving their relative order. Tom proceeds and produces the following implementation.

```
public int popMax()
{
  Contract.Requires(!isEmpty());
  int m = elems[0];
  for (int i = 1; i < next; i++)
    if (elems[i] > elems[m])
      m = i;
  int ret = elems[m];
  for (i = m; i < next - 1; i++)
    elems[i] = elems[i + 1];
  return ret;
}
```

Since a maximum element needs to be returned, the operation is enabled only if the Stack instance is not empty. On a first pass, the position of the maximum element is stored in m. The elements to "the right" of m are shifted so that they remain in the same order, but occupying m's position.

Tom is confident with his implementation, but DataXtructures Inc. mandates that every new functionality needs to be subject to unit testing, so he writes the following test.

```
void testPopMax()
{
  Stack s = new Stack(10);
  s.push(3); s.push(42);
  s.push(1); s.push(17);
  int max = s.popMax();
  Contract.Assert(max == 42);
}
```

This test is executed and fortunately it provides a 100% branch coverage on the popMax implementation. Furthermore, as Tom expected, the test passes and he can now focus on other assignments. At that point, Tom remembers that he can generate an new EPA that features the popMax operation.

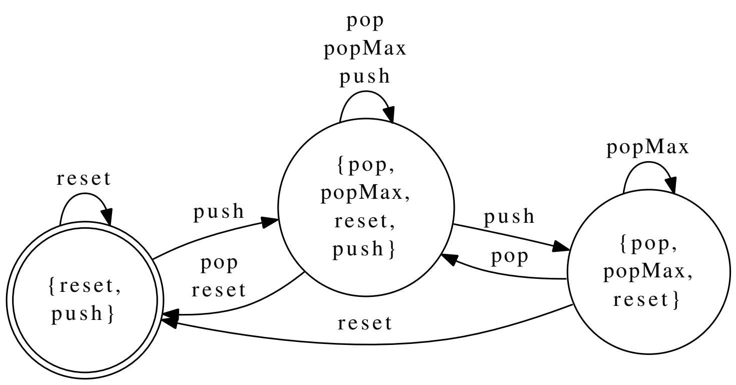

Fig. 4. EPA for the `Stack` class with `popMax` action

Tom then gets the EPA in Figure 4. As expected, the abstract states are similar to the ones in the previous EPA (see Figure 3). The `popMax` operation is enabled whenever the `pop` operation is enabled, since they indeed have the same requires clause.

After checking the abstract states, Tom decides to look at the transitions. Particularly, he is interested in the newly added `popMax` operation. He soon notices that there is a `popMax`-labeled transitions looping over the {`pop`, `popMax`, `reset`} abstract state. Since pushing is not allowed on this state, it represents full stacks. Popping elements with the standard `pop` operation on this state takes the stack back to the 'half-full' state. However, using the newer `popMax` operation leaves the stack full. The target of the `popMax` transition should be the 'half-full' state, so something looks suspicious. Similarly, there seems to be a missing `popMax`-labeled transition that takes a 'half-full' stack back to the initial empty state.

Tom figures out that since the `popMax` operation passed his test, the problem is not related to the returned element, but to the state in which the `Stack` structure is left after the operation. In particular, the fact that the only `popMax`-labeled transitions are loops indicates that the size of the structure appears to be unchanged by the operation, when it should be decreasing.

Tom reviews the implementation of `popMax` and discovers that the `next` variable is not altered and this is a bug! He then adds a `next--;` operation right before the end of the implementation in order to fix this.

Contractor is invoked once more after Tom has fixed the bug and the resulting EPA is shown in Figure 5. The set of abstract states remains the same, but the awkward `popMax` loop over the rightmost abstract state is now gone. Furthermore, Tom notices that the `popMax` operation presents the same (abstract) behaviour as `pop`.

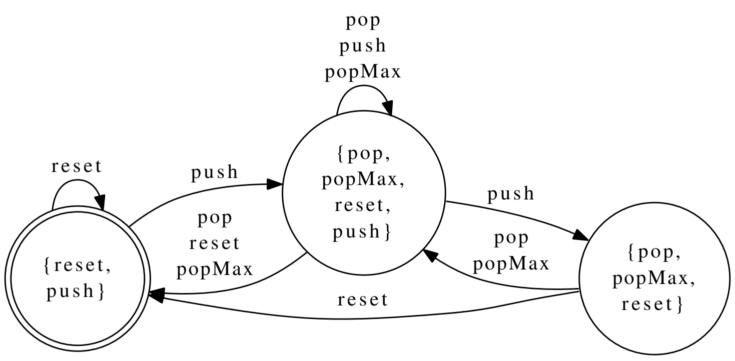

Fig. 5. EPA for the `Stack` class with fixed `popMax` action

3.3 Refining the EPA States

Tom is happy that he finished his first assignment, but his colleagues are concerned that the `popMax` operation is rather inefficient. Apparently, `Stack` instances are increasingly used in contexts on which the elements are orderly pushed in ascending order.

In such scenarios, going through all the elements looking for the maximum is unnecessary since returning the last element is sufficient. The problem is that returning the last element is not valid if the `Stack` instance is not ordered. Tom figures out a strategy to keep track of whether the elements are ordered or not by introducing an additional class field, as follows.

```
public class Stack {
  private int[] elems;
  private int next;
  private int sorted;

  [ContractInvariantMethod]
  private void Invariant()
  {
    Contract.Invariant(0 <= next);
    Contract.Invariant(next <= elems.Length);
    Contract.Invariant(0 <= sorted);
    Contract.Invariant(sorted <= next);
    Contract.Invariant(elems.Length > 1);
  }

  // ...
}
```

The `sorted` field stores the length of the largest sorted prefix in the `elems` array. If `sorted` is equal to `next` then the `Stack` instance is sorted and returning the last element accounts for returning the maximum. Otherwise, a linear scan is still necessary.

Notice that the Stack invariant is extended to indicate that sorted is in range. The invariant could also be extended to indicate that sorted actually marks the size of the biggest sorted prefix, but this weaker invariant is enough for Tom's purposes.

The operations responsible of pushing and popping elements need to carefully update the new sorted field, as follows.

```
public void push(int k)
{
  Contract.Requires(!isFull());
  if (isEmpty() || sorted == next && k >= top())
    sorted++;
  elems[next] = k;
  next++;
}

public void pop() {
  Contract.Requires(!isEmpty());
  if (sorted == next)
    sorted--;
  numberOfElements--;
}
```

Having extended the Stack representation and operations to work with the extra field, Tom re-implements the popMax operation with the optimisation.

```
public int popMax()
{
  Contract.Requires(!isEmpty());
  if (sorted == next) {
     sorted--;
      return elems[next];
  } else {
    int m = elems[0];
    for (int i = 1; i < next; i++)
      if (elems[i] > elems[m])
        m = i;
    int ret = elems[m];
    for (i = m; i < next - 1; i++)
      elems[i] = elems[i + 1];
    next--;
    return ret;
  }
}
```

If the Stack instance is sorted, the last element is returned. Otherwise, the implementation is the same as before.

Tom first runs the unit test that he already had from the unoptimised version and it passes. He then runs CONTRACTOR to get a new EPA, which is shown in Figure 6.

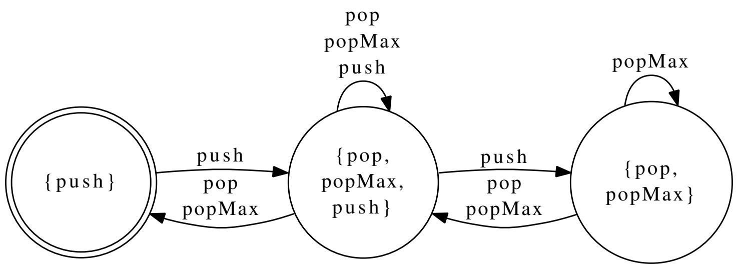

Fig. 6. EPA for the `Stack` class with optimized `popMax` action

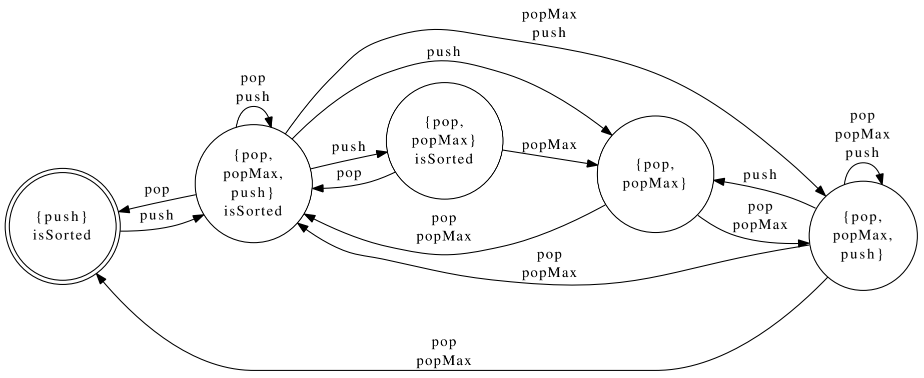

Fig. 7. Refined EPA for the `Stack` class with optimized `popMax` action

First of all, notice that for presentation purposes the `reset` operation is now not considered when building the EPA. The `popMax` operation appears to be doing its job since it is accompanying every `pop` transition. The problem is that there is an extra `popMax`-labeled transition looping over the full state.

Since the unoptimised version worked and the optimised seems to have a bug, Tom decides to get a finer-grained abstraction and check what is wrong with his code. Fortunately, CONTRACTOR provides the user with the ability to add additional predicates that can be used to refine the set of abstract states.

Tom adds the predicate `sorted == next`, which he names `isSorted`, for presentation purposes. The refined EPA that he gets is depicted in Figure 7.

There are now 5 abstract states, from left to right:

– The initial state, which is sorted. An empty array can not be unordered.
– The sorted 'half-full' state.
– The sorted full state.
– The unsorted full state.
– The unsorted 'half-full' state.

Pushing from the initial state takes the `Stack` instance to the sorted ‘half-full’ state; having a single element the elements have to be sorted.

Pushing from the sorted ‘half-full’ state is non-deterministic, as it can go to:

- Itself. In this case, the pushed element is higher than the top of the `Stack` instance, and therefore it remains sorted. There is also still room for more elements.
- The sorted full state. Similar to the previous case, but with no room for more elements.
- The unsorted full state. The pushed element is lower than the top of the `Stack` instance, so it is no longer sorted. There is no more room for new elements.
- The unsorted ‘half-full’ state. Similar to the previous case, but with extra room for other elements.

On the other hand, pushing a new element on the unsorted ‘half-full’ state (the one in the far right) is still non-deterministic, but it can never result in the `Stack` instance getting sorted.

The `pop` operation behaves dually. It can make an unsorted `Stack` become sorted. On sorted stacks, popping will never mangle the elements.

Tom expects the `popMax` operation to behave as `pop`. When the `Stack` is unsorted they share every transition. However, when the `Stack` is sorted, `popMax` behaves oddly. For instance, there is a missing `popMax`-labeled transition from the sorted ‘half-full’ state going back to the initial one. This manifestation is similar to the bug that Tom discovered in his original implementation.

On top of that, there are `popMax`-labeled transitions going out of any sorted abstract state to its unsorted counterpart. This is suspicious since taking elements out of an ordered array should never result in their elements getting unordered.

With this information in hand, Tom decides that his new implementation is working fine on unsorted stacks, but his optimisation to deal with sorted stacks is buggy. He then discovers that he forgot (again!) to update the `next` field in the case in which the elements are ordered (the `then` branch).

Tom then fixes the bug and runs Contractor once more to confirm that the suspicious elements in the original abstraction are now gone.

3.4 Refining the EPA Transitions

Having finished his previous assignment, Tom can now focus on his second task. Some `Stack` users need to remove several elements at once. Tom has to implement a `popN(int n)` operation, which takes an integer `n` and removes that amount of elements from the top of the `Stack` instance. If there are fewer than `n` elements, then the instance remains empty and the actual number of popped elements is returned.

Tom is eager to implement this new method, and in a couple of minutes he gets the following code.

```
public int popN(int n)
{
  Contract.Requires(!isEmpty());
  Contract.Requires(n >= 1);
  for (int i = 0; i < n; i++) {
    pop();
    if (isEmpty())
      break;
 }
 return i;
}
```

Similarly to the previous pop operations provided by the `Stack` class, `popN` requires the instance to have at least one element. Furthermore, the amount of elements to be popped has to be at least 1.

Tom decides to create a simple unit test to see what happens in two scenarios: *(a)* when the requested amount of pops can be fulfilled; and *(b)* when it is greater than the amount of elements in the stack.

```
1 void testPopN()
2 {
3   Stack s = new Stack(10);
4   s.push(99); s.push(89); s.push(79);
5   int n1 = s.popN(1);
6   Contract.Assert(n1 == 1);
7   int n2 = s.popN(3);
8   Contract.Assert(n2 == 2);
9 }
```

The `Assert` in line 6 checks for scenario *(a)*, while the one in line 8 checks for scenario *(b)*.

Fortunately, once again, the test achieves a 100% branch coverage over the added functionality. Furthermore, the first `Assert` passes correctly, so the operation seems to work properly when there are enough elements. Unfortunately, the test case fails to pass the second `Assert`. The amount of elements in the `Stack` instance at the time of the second `popN` operation is 2. However, when attempting to pop 3 elements, the `popN` operation returns 1 instead of 2.

As usual, Tom decides to construct an EPA of the `Stack`, which we can see in Figure 8. Unfortunately, the bug does not seem to be reflected in the abstraction. All the `popN`-labeled transitions behave like `pop`, with the exception that `popN` can also go directly from the full abstract state back to the initial one.

In his previous assignment, Tom used Contractor to refine the EPA states. Similarly, Contractor features the possibility to refine abstract transitions too. In order to do so, Tom needs to specify which label he wants to refine, and what conditions he wishes to use.

In this particular case, Tom needs to refine the `popN`-labeled transitions using two conditions:

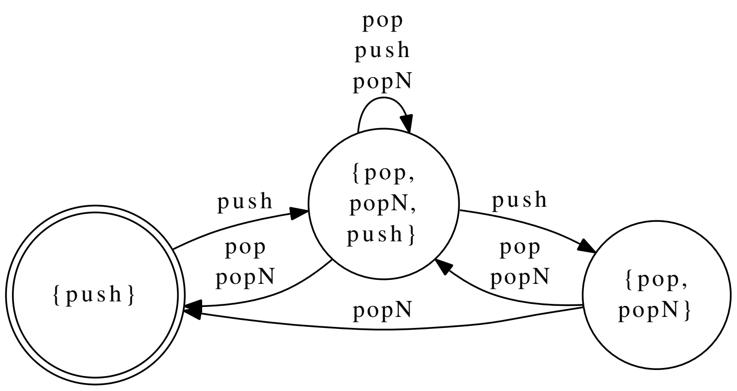

Fig. 8. EPA for the `Stack` class with `popN` action

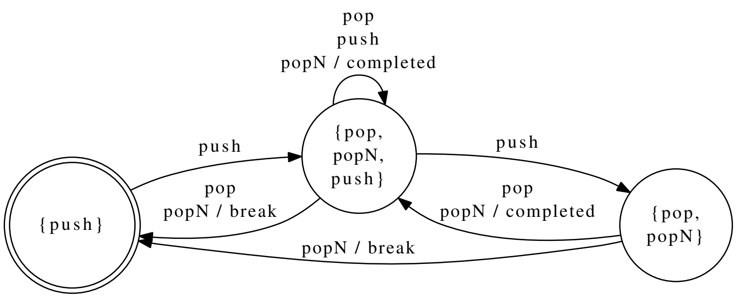

Fig. 9. EPA with refined transitions for the `Stack` class with `popN` action

`completed.` This is the case on which the amount of popped elements equals the amount that the user asked. Formally, `n == popN(n)`.

`break.` In this other case the operation has to stop early since there are no more elements to pop. Formally, `n < popN(n)`.

The refined EPA that Tom gets using this feature is depicted in Figure 9. Notice that now the `popN` transitions show extra information indicating which of the two conditions holds on each case.

The `popN / break` transitions always return to the initial abstract state. This seems fine to Tom, since in those cases the operation had to stop early due to lack of elements. On the other hand, the `popN / completed` transitions always go to the 'half-full' abstract state. This is suspicious, since a user could ask `popN` to return exactly the amount of elements currently in the stack, which should empty the instance.

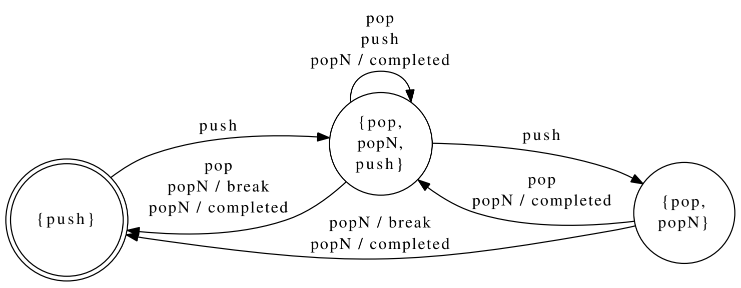

Fig. 10. EPA with refined transitions for `Stack` with the fixed `popN` action

Tom decides to write a new test case in order to figure out what is wrong with his implementation.

```
void anotherTestPopN()
{
  Stack s = new Stack(10);
  s.push(99); s.push(89);
  int n = s.popN(2);
  Contract.Assert(n == 2);
}
```

Tom does a step-by-step execution and discovers that when the second element is popped, the `Stack` instance is empty at that moment. Consequently, the `if` statement in line 7 of the `popN` implementation is taken. Therefore, the execution of the `popN` operation is halted before the `i` variable is incremented.

The bug can be solved by using a `while`-loop instead of a `for`-loop, as follows.

```
public int popN(int n)
{
  Contract.Requires(!isEmpty());
  Contract.Requires(n >= 1);
  int i = 0;
  while (i < n) {
    pop();
    i++;
    if (isEmpty())
      break;
  }
  return i;
}
```

With this fixed version, the EPA in Figure 10 looks much better. Tom's job for the day is completed!

4 Validation Guidelines

The previous section showed a fictional story on which EPAs where used to identify and locate problems in a software artefact. Based on our experience in various real-life software artefacts [8,9], we now present a series of guidelines that developers can use as heuristics that aid identification of “suspicious behaviour” during the validation process.

We organise the heuristics into two categories. The first category is of a more semantic nature while the second is related to the structure of the EPA.

We hypothesise that one of the benefits of the approach presented is that the level of abstraction defined by the enabledness criterion is intuitive and modelers can interpret the different abstract states into the problem domain with relative ease. The first two heuristics we developed confirm, to some extent, this hypothesis.

– **Understanding states.** There are certain abstract states in the EPA that can be easily interpreted to particular situations of the system under analysis. Identifying empty, half-full and full abstract states in any of the `Stack` EPAs is an example of this.

  When it is not possible or not easy to associate a particular states with a declarative description of the set of instances that it abstracts, this may be an indication that there is a problem with the program under analysis. We have found that in these cases, it was often the case that the state should have been inconsistent (and hence should not have appeared in the EPA) but that the requires clauses of enabled actions or the system invariant were (incorrectly) too weak. This is the case in Figure 2, in which the invariant for the system was `true`.
– **Understanding action sequences.** On the other hand, states which can be declaratively traced to a meaningful set of instances are good candidates for analysing action sequences. Following fragments of traces from these states may lead to discovering a certain sequence of actions which should not be allowed by the program. Programmers should be aware that, given the approximate nature of the abstraction, the appearance of a (non singleton) trace is not a guarantee that it denotes a feasible action sequence.

  An example of this strategy is what led to the discovery that the EPA in Figure 8 lacks `popN / completed` transitions going back to the initial state.

We also identified the following *structural characteristics of an EPA* that can help pinpoint problems in the program under analysis:

– **Large state space.** A large state space in the EPA may be an indication of either a poorly designed set of operations. The intuition is that a set of operations that are intended to be used together to provide a more complex service (e.g., a protocol, a public API) will conceptually have a few modes that characterise the set operations available at a given moment. An unmanageable set of enabledness states is an indication that the protocol, class or API is either extremely complex to be used or that it is incorrectly implemented. More specifically, a large state space can be an indication of problems with requires clauses. A good strategy is to question why different states in the EPA differ in the actions that they enable.

An example of this problem is showcased in the MS-WINSRA case study in [8].

- **Deadlock states.** The presence (or absence) of a deadlock state is something that should be analysed in detail when validating a program using EPAs. By definition of EPA there can be only one deadlock state, the state whose action set is empty. The presence of an unintended deadlock state in an EPA is likely to be an indication of a bug in the actions that evolve into that state.

  The MS-NSS case study in [8] presents a deadlock state which is studied and validated.
- **Sink states.** Similarly to deadlock states, states which only have outgoing transitions leading back to it can be indicators of problems. They are very similar to deadlock states since they indicate that once this "operation mode" is reached it can not be abandoned.

  For instance, the `Stack` EPA in Figure 2 presents a suspicious sink state.
- **Missing action.** If a given specified action is not present in any of the EPA reachable states then this is an indication that something is not quite right. It may be the case that the requires clause for that action is inconsistent when combined with the system invariant. It might also be the case that none of the other actions leave the system in a state which enables the missing action.
- **High fan-in.** States in an EPA that have a large number of incoming transitions can be an indication of problems. In particular, they are typically undesirable since they cause history loss for all the paths that reach the state. These states can be an indication of problems in requires clauses that when corrected end up partitioning the high fan-in state into several states.

  The MS-WINSRA case study in [8] presents this problem due to weak requires clauses.
- **Highly non-deterministic actions.** When a state has a large number of outgoing transitions labeled with the same action it is usually symptomatic of a problem. Such situations may be caused by two different scenarios. Firstly, it may be the case that the action is intrinsically non-deterministic. If this is the case, it can be a symptom that this action is a good candidate to be tested under different scenarios in order to trigger/cover all of its behaviour space.

  Secondly, a highly non-deterministic action on an abstract state can also happen if the predicate for the state is weak. For instance, an action that updates the system following the formula $(A_1 \Rightarrow B_1) \wedge \ldots \wedge (A_n \Rightarrow B_n)$ may generate undesired non-deterministic behaviour in a state where A_i holds for several values of i. In these cases, it may be the case that a requires clause or the system invariant requires strengthening.

  The {`pop, popMax, push`} `isSorted` abstract state in Figure 7 presents a highly non-deterministic `push` operation.

– **Mirrored actions.** If whenever there is a transition labeled with a given action a_1, there is another transition with the same origin and destination state labeled with action a_2, this is an indication that both actions were specified independently but are treated in the same way by the system. It may be the case that one action was copied from the other but the programmer forgot to modify the appropriate differences between the two (known as copy-paste bugs).

  This is the case with `pop`—`popMax` in Section 3.3, as well as `pop`—`popN` in Section 3.4.

Some of the heuristics presented in this section are straightforward to implement as a feature in our Contractor tool, and in fact some of them are already implemented. These include the detection of deadlock or sink states, mirrored or missing actions or enabled actions with missing transitions.

5 Closing Remarks

In this article we have shown how abstractions can play a key role in various development activities. More concretely, we presented enabledness-preserving abstractions and their application in a fictional development story involving code understanding, the addition of new functionality and the refactoring of parts of the implementation.

Even when this story is based on a toy stack implementation, EPAs have shown to provide useful information on various industrial scale software artifacts. For instance, in [8] we show the application of our Contractor tool to 2 Microsoft protocol descriptions in the form of pre/postcondition contracts with up to 33 actions. Our tool scaled well, keeping the construction time below 4 minutes in a standard desktop computer. More importantly, the resulting EPA led to the discovery of flaws in the descriptions. In [9] we apply Contractor directly on the Java Development Kit (JDK) 1.4 implementation of various standard classes such as `ListItr` (List iterator) or `PipedOutputStream`. The running times ranged from 8 seconds to 5 minutes, and the EPAs led to discoveries such as undocumented legal behaviour with respect to the official documentation.

Based on this experiences, this paper also presents a series of guidelines and checklists that can guide programmers in their use of EPAs as a visual aid to find suspicious elements in their programs.

While Contractor constructs EPAs statically and automatically, it requires the developer to identify and annotate important program elements such as requires clauses. The problem of tackling the annotation burden has been widely studied, and different approaches have been proposed which range from type inference or dataflow analysis (e.g., [17]) to carefully choosing default values [10]. More concretely, Code Contracts [3] presents an outstanding ability to infer requires clauses as the ones used in Section 3.

Enabledness-based abstractions constitute a good example of abstractions aimed at user inspection and validation. While we found EPAs useful in our experience, we believe that features such as state refinement via extra predicates (Section 3.3) or transitions refinement (Section 3.4) present a good opportunity to let developers "zoom in and out" the abstraction in order to fit their needs. That said, we envision that other carefully chosen non enabledness-based levels of abstraction could potentially help developers in their activities as well.

Acknowledgements. The work reported herein was partially supported by CONICET, UBACyT 2011-813 and 20020090300064, PIP112-200801-00955KA4, PICT-PAE 37279, ERC StG PBM-FIMBSE and PICT 2010-2351.

References

1. Alur, R., Černý, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: POPL 2005, pp. 98–109 (2005)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
3. Andersen, M., Barnett, M., Fahndrich, M., Grunkemeyer, B., King, K., Logozzo, F., Patel, V., Zuniga, D.: Code Contracts (2009), `http://research.microsoft.com/en-us/projects/contracts/`
4. Beckman, N., Nori, A.: Probabilistic, modular and scalable inference of typestate specifications. In: PLDI (2011)
5. Beschastnikh, I., Brun, Y., Sloan, S., Ernst, M.: Leveraging existing instrumentation to automatically infer invariant-constrained models. In: FSE 2011 (2011)
6. Beyer, D., Henzinger, T., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9, 505–525 (2007), `http://www.springerlink.com/index/10.1007/s10009-007-0044-z`
7. Bierhoff, K., Aldrich, J.: Plural: checking protocol compliance under aliasing. In: ICSE, pp. 971–972. ACM (2008)
8. de Caso, G., Braberman, V., Garbervetsky, D., Uchitel, S.: Automated abstractions for contract validation. IEEE Transactions on Software Engineering 38(1), 141–162 (2012)
9. de Caso, G., Braberman, V.A., Garbervetsky, D., Uchitel, S.: Program abstractions for behaviour validation. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, pp. 381–390 (2011)
10. Chalin, P., James, P.R.: Non-null References by Default in Java: Alleviating the Nullity Annotation Burden. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 227–247. Springer, Heidelberg (2007)
11. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: Workshop on Dynamic Systems Analysis 2006 (2006)
12. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: ISSTA 2010 (2010)
13. DeLine, R., Fahndrich, M.: Enforcing high-level protocols in low-level software. In: PLDI 2001, pp. 59–69 (2001)
14. Demsky, B., Rinard, M.: Automatic extraction of heap reference properties in object-oriented programs. IEEE Transactions on Software Engineering 35, 305–324 (2009)

15. Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69, 35–45 (2007), http://linkinghub.elsevier.com/retrieve/pii/S016764230700161X
16. Esparza, J.: Decidability of model checking for infinite-state concurrent systems. Acta Informatica 34, 85–107 (1997), http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/s002360050074
17. Flanagan, C., Leino, K.: Houdini, an Annotation Assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
18. Gabel, M., Su, Z.: Symbolic mining of temporal specifications. In: ICSE 2008, pp. 51–60 (2008), http://portal.acm.org/citation.cfm?id=1368096
19. Ghezzi, C., Mocci, A., Monga, M.: Synthesizing intensional behavior models by graph transformation. In: ICSE 2009, pp. 430–440 (2009)
20. Giannakopoulou, D., Păsăreanu, C.S.: Interface Generation and Compositional Verification in JavaPathfinder. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 94–108. Springer, Heidelberg (2009)
21. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
22. Grieskamp, W., Kicillof, N., MacDonald, D., Nandan, A., Stobie, K., Wurden, F.: Model-based quality assurance of Windows protocol documentation. In: ICST 2008, pp. 502–506 (2008), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4539580
23. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Transactions on Software Engineering and Methodology (TOSEM) 5(3), 231–261 (1996)
24. Henzinger, T., Jhala, R., Majumdar, R.: Permissive interfaces. In: ESEC/FSE 2005, pp. 31–40 (2005)
25. IEEE: IEEE Standard Glossary of Software Engineering Terminology (September 1990)
26. Kramer, J.: Is abstraction the key to computing? Commun. ACM 50, 36–42 (2007), http://doi.acm.org/10.1145/1232743.1232745
27. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
28. Lee, D., Yannakakis, M.: Online minimization of transition systems (extended abstract). In: STOC 1992, pp. 264–274 (1992), http://portal.acm.org/citation.cfm?doid=129712.129738
29. Liu, L., Meyer, B., Schoeller, B.: Using Contracts and Boolean Queries to Improve the Quality of Automatic Test Generation. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 114–130. Springer, Heidelberg (2007)
30. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: ICSE 2008, pp. 501–510 (2008)
31. Nanda, M., Grothoff, C., Chandra, S.: Deriving object typestates in the presence of inter-object references. ACM SIGPLAN Notices 40(10), 77–96 (2005)
32. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
33. Pradel, M., Gross, T.R.: Automatic Generation of Object Usage Specifications from Large Method Traces. In: ASE 2009, pp. 371–382. IEEE (November 2009), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5431756

34. Sasnauskas, R., Dustmann, O.S., Kaminski, B.L., Wehrle, K., Weise, C., Kowalewski, S.: Scalable symbolic execution of distributed systems. In: Proceedings of the 2011 31st International Conference on Distributed Computing Systems, ICDCS 2011, pp. 333–342. IEEE Computer Society, Washington, DC (2011), http://dx.doi.org/10.1109/ICDCS.2011.28
35. Strom, R., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE TSE 12(1), 157–171 (1986)
36. Uribe, T.: Abstraction-based Deductive-algorithmic Verification of Reactive Systems. Stanford University, Dept. of Computer Science (1999)
37. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
38. Zoppi, E., Braberman, V., de Caso, G., Garbervetsky, D., Uchitel, S.: Contractor.net: inferring typestate properties to enrich code contracts. In: Proceedings of the 1st Workshop on Developing Tools as Plug-ins, TOPI 2011, pp. 44–47. ACM, New York (2011), http://doi.acm.org/10.1145/1984708.1984721