Tesis Doctoral

Modelos abstractos de comportamiento basados en habilitación

de Caso, Guido

2013

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Modelos abstractos de comportamiento basados en habilitación

Tesis presentada para optar al título de Doctor de la Universidad de Buenos Aires
en el área Ciencias de la Computación

Guido de Caso

Director de tesis:	Dr. Sebastián Uchitel
Consejero de estudios:	Dr. Víctor Braberman

Buenos Aires, 2013

Modelos abstractos de comportamiento basados en habilitación

Resumen: Muchas interfaces programáticas de aplicación (APIs) presentan restricciones no triviales respecto al orden en que sus operaciones deben ser invocadas. Para los desarrolladores a cargo de implementar dichas APIs, validar si las mismas proveen el comportamiento esperado es un problema desafiante. De todas formas, incluso en la ausencia de requerimientos formales, los desarrolladores de APIs poseen un *modelo mental* informal sobre el comportamiento esperado de la API. Este trabajo apunta a asistir a estos desarrolladores en la validación de sus APIs mediante la construcción de modelos que puedan comparar con sus modelos mentales. Presentamos las abstracciones basadas en habilitación (EPAs), un novedoso modelo de comportamiento de grano grueso que presenta una versión sobreaproximada del protocolo de uso de una API. Las EPAs agrupan las instancias concretas de una API que habilitan el mismo conjunto de operaciones, lo cual ofrece buena trazabilidad entre el modelo y la API. Brindamos algoritmos que construyen EPAs a partir de especificaciones o implementaciones de APIs. Luego estudiamos nuestro enfoque mediante una serie de casos de estudio en los cuales expertos de dominio usaron EPAs para identificar problemas en APIs de escala industrial, una evaluación de la expresividad de las EPAs y tres experimentos controlados apuntando a establecer cómo entienden los desarrolladores a las EPAs. Estas experiencias confirman que los modelos de grano grueso tales como las EPAs pueden jugar un rol importante en procesos manuales tales como validación.

palabras clave: abstracciones de comportamiento de grano grueso, modelos diseñados para la validación guiada por humanos, protocolo de uso de APIs, guías para la validación, expresividad vs. entendibilidad.

Enabledness-based abstract behaviour models

Abstract:

Many *application programming interfaces* (APIs) present non-trivial restrictions with respect to the order in which their operations ought to be called. For a developer in charge of implementing an API, validating whether it provides the expected behaviour is a challenging problem. Nevertheless, even in the absence of formal requirements, API implementers possess an informal *mental model* of the expected API behaviour. This work aims to assist these developers in the validation of their APIs by constructing models that they can compare against their mental models. We introduce *enabledness-preserving abstractions* (EPAs), a novel *coarse-grained* behaviour model which presents an overapproximated version of an API usage protocol. EPAs group concrete instances of an API that enable the same set of operations, offering good traceability links between the model and the API. We present EPA construction algorithms from either API specifications or API implementations. We then study our approach by means of a series of cases studies where experts used EPAs to identify issues in industrial strength APIs, an evaluation of EPA expressiveness, and three controlled experiments aimed at establishing how developers understand EPAs. These experiences confirm that coarse-grained models such as EPAs can play an important role in human-intensive processes such as validation.

keywords: coarse-grained behaviour abstraction, models aimed at human-driven validation, API usage protocol, validation guidelines, expressiveness vs. understandability.

Agradecimientos

Primero quiero agradecer a Sebastián y a Víctor por guiarme en este proceso tan interesante que es el de avanzar en terrenos inexplorados. Gracias por generar un clima de confianza, respeto y colaboración que me permitió hacer mis primeras armas en el campo de la investigación.

Este trabajo no hubiera sido posible de no ser por Diego, quien siempre nos acompañó desde un primer momento, desempeñando el rol de colaborador de lujo. Tanto en lo académico como en lo personal Diego me ayudó a superar los innumerables desafíos que un doctorado supone.

Mis compañeros de oficina hicieron que el trabajo realizado estos años haya sido de lo más entretenido y llevadero: Dani, Dipi, Esteban, Ferto, Herno, Sherman y Sherwood. ¡Grosos! Quiero agradecer también al resto de los miembros del grupo LaFHIS.

Quiero hacer una mención especial a Yuri Gurevich, Nikolaj Bjørner y el resto de la gente de Microsoft Research, quienes me apoyaron durante mi doctorado y me permitieron colaborar con ellos en proyectos de lo más interesantes.

Quiero agradecer a mi familia por todo el apoyo y por haberme inculcado los valores del estudio, el esfuerzo y la autosuperación.

Para Celes, que en estos años demostró ser poseedora de paciencia y dulzura infinitas, no tengo más que decirle, una vez más, que lo que siento por ella es amor que tiende a ∞.

Contents

Part I

Prelude

CHAPTER 1

Introduction

Verification and validation are artefact evaluation activities carried out at 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 natures.

Verification. Verification aims at determining whether an artefact satisfies specific properties [IEE90]. 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 variety of tools such as model checkers [BHJM07], theorem provers [NPW02], simulators [KNP11] or symbolic executers [SDK+11]. There are, of course, both theoretical (indecidability results, e.g., [Esp97a]) and practical (e.g., state explosion [Val98]) 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.

Validation. Validation 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 it. 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 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 [ACH+95], as well as internal consistency and satisfiability in requirements specifications [HJL96]). 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 validation task, so eventually human intervention is required.

Application Programming Interfaces (APIs). In this thesis we are particularly interested in the validation of software artefacts that have non-trivial requirements with respect to the order in which their methods or procedures ought to be called. Such is the case for many *application programming interfaces* (APIs) [BKA11]. While they appear everywhere in the form of frameworks, public class interfaces or web services, previous work has shown that developers struggle to learn how to use APIs. Furthermore, in practice, descriptions of the intended behaviour for APIs 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 behaviour models [SY86] from API implementations which are then used to verify if client code conforms to the implemented protocol [AČMN05, DKM+10]. Such approaches address only part of the problem: they assume the code from which the behaviour model 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 assisting API implementers to validate if their APIs provide the intended behaviour when descriptions of this

behaviour are informal, partial or non-existent. In this context, approaches such as testing [GKSB11] or verification [CK05] are inapplicable due to the absence of an oracle, or target desirable properties, respectively.

However, we believe that even in the absence of a full-fledged formal specification, API developers still have an informal understanding of the class they are building and the desired requirements that it has to meet. We usually refer to this informal understanding as a *mental model*.

Our working hypothesis is that, presented with an automatically inferred model comparable to his mental model, a developer can reason about an API and gain insight.

The basic idea behind model inference (or synthesis) is to automatically analyse, either statically (e.g, [AČMN05, HJM05]) or dynamically (e.g., [GMM09, EPG+07]), a software artefact and produce an abstract description of it. The resulting models are sometimes used as input for other techniques [BN11]. However, it is often the case that the inferred models are intended to be directly consumed by developers [BBSE11].

We are interested in the latter scenario. For models to be consumed by humans, they have to convey relevant aspects about the original artefact in a fashion amenable to human comprehension. This is a balancing act between expressiveness and understandability. If a model conveys too much information it may overwhelm the developer. If it is too simple it is unlikely to be of assistance.

In the vast majority of literature in this area, evidence of developer understanding of such models is often missing or anecdotal [CZvD+09, TTDBS07], with no statistical relevance. Moreover, few user studies have been conducted, mostly with negative results. For instance, in the context of models in the form of likely invariants, a recent user study [SHKR12] has shown that developers struggle to correctly understand them.

Abstraction. When inferring models from APIs, abstraction is paramount. 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 [Kra07], 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.

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 [SY86, DF01, NGC05, BN11] or interfaces [AČMN05, GP09, HJM05]. The aim is to statically obtain finite state machine representing a *safe* model from a client perspective, using techniques such as automata learning [AČMN05, GP09, HJM05] or abstract interpretation [NGC05].

The safety requirement associated with 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 [GP09, HJM05] 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., [BA08]). 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 [Uri99, GS97]. 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., [LY92, GS97, LMS07, GKM+08a]). For instance [GS97] 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, [LMS07] 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., [DLWZ06, GMM09, GS08, LMP08, DKM+10, BBSE11, PG09]). 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, [GS08] produces an automata by collecting information from the client's actual usage of a set of operations (underapproximation). Adabu [DLWZ06] 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 their sign), leading to both under and overappproximation of the concrete state. Other approaches [GMM09, LMP08] use invariant detection tools such as Daikon [EPG+07] in order to generalise the set of traces and obtain more conservative models.

Abstractions for validation. We are interested in 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 of choosing the abstraction.

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

For instance, the approach followed in [BBSE11] uses logging mechanisms already in place and regular expressions to obtain behaviour models almost without user intervention. The logs are mined 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 [DR09]. The 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 overapproximations are better suited for validation since they are capable of exhibiting all the potential behaviour of the artefact.

In other words, during a validation task, the developer does not necessarily have a specific property to be verified in mind. In that context, underapproximated abstractions may omit relevant aspects of the behaviour. On the other hand, overapproximated abstractions display all possible legal behaviour, which is more suitable when the developer is exploring an artefact during a validation scenario. Of course, with overappproximation comes the potential pitfall of conservatively displaying too much information and confound the user. In this work we show a particular kind of overapproximated abstraction that to be informative enough to unearth interesting

properties from the original artefact, and yet concise enough to prevent overwhelming developers.

Enabledness Preserving Abstractions (EPAs). In this work we introduce and discuss *enabledness-preserving abstractions* (or EPAs) [dCBGU12a, dCBGU11], a novel *coarse-grained* behaviour model designed to assist developers in the validation of APIs. EPAs convey a concise abstract representation of an API's usage protocol and are intentionally designed to be understandable, sometimes at the cost of sacrificing expressiveness.

The key idea behind the abstraction mechanism is to group those concrete instances of an API that enable the same set of operations. This abstraction level has proved to offer good traceability between elements in the inferred model and the original artefact. Furthermore, EPAs are compact enough to be human-understandable, and yet they are rich enough to assist developers in key tasks such as validation.

As we stated above, a model is safe [AČMN05] if no call sequence violates the library's internal invariants; it is permissive if it contains every such safe sequence. Previous approaches have aimed (e.g., [HJM05]) at modular program analysis using models 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 EPAs are permissive at the cost of safety. As EPAs are intended to be compared to mental models, failing to offer all of the legal behaviour (as it may occur in the case of safe models) may hinder the validation process.

In this work we present construction algorithms that take an API as input and statically and automatically produce an EPA. The input API can consist of either a pre/postcondition specification for each action, or a fully-fledged source code implementation. We implemented multi-threaded versions of these algorithms and released an open-source tool called Contractor.

In order to support our claims about EPAs usefulness, we validated our approach in various ways. Firstly, we analysed whether our algorithms are efficient enough to deal with real-life APIs.

Secondly, we conducted a series of case studies where expert reviewers used EPAs to guide the validation process of a series of industrial-strength APIs. These case studies led to the identification of issues in the APIs such as undocumented behaviour or ambiguities in the requirements. Based on this experience, we enumerated a list of EPA validation guidelines. These guidelines can help developers or reviewers to identify suspicious EPA elements, which can in turn lead to the identification of problems in the input API.

Then, we studied EPAs expressiveness and understandability. We analysed their expressiveness by conducting an evaluation of how sensitive they are to the presence of defects in the source code of the API implementation. Regarding understandability, we conducted a user study where developers were presented with an EPA that was either obtained from the original API implementation, or from a defective version of it. Developers were asked to identify whether or not these EPAs matched their mental model of the expected API behaviour. The overall goal of this study was twofold: to determine user detection effectiveness for EPAs that were generated from defective versions of the source code; and to understand what factors lead to successful or unsuccessful detection of these EPAs.

1.1. Contributions

The contributions of this thesis are:

- The idea of using overapproximated abstractions for validation of APIs.
- The definition of *enabledness-preserving abstractions* (EPAs), a novel coarse-grained behaviour model aimed at human inspection.
- Algorithms to automatically and statically construct enabledness-preserving behaviour models from either
  - API pre/postcondition specifications
  - source code implementations accompanied with invariants and requires clauses.
- The implementation of these construction algorithms into a publicly-available tool named Contractor.
- The study of how EPAs are used by domain experts to validate a series of industrial strength API specifications and implementations on which issues were found.
- A list of validation guidelines that can help developers identify suspicious elements in an EPA.
- An evaluation of EPA expressiveness by means of analysing how sensitive they are to the presence of defects in the input artefact.
- An analysis of how well developers understand EPAs by means of a series of controlled user studies.

To the best of our knowledge, we present the first approach that given an API implementation or specification, statically and automatically constructs a model that accepts a superset of the legal API traces and is therefore suitable for human inspection.

1.2. Limitations

The results presented in this work depend on a series of assumptions:

- A part of our efforts are aimed at generating EPAs from pre/postcondition specifications. It is a fact that these kind of formal specifications are seldom found in practice.
- We also support the analysis of source code, which is a much more commonplace software artefact. However, in order to produce EPAs for a program we require it to be equipped with requires clauses for each of its operations as well as a system (or class) invariant.
- Furthermore, as we discuss in Chapter 4, the provided requires clauses for each action need to be splittable into 2 parts: one that predicates over the action parameters and the other over the system variables. If this requirement is not met, the technique supports the use of requires clauses over and underapproximations.

- The nature of EPAs makes them good candidates to help the user discover and understand the usage protocol of a software artefact. When dealing with programs (or specifications) that feature a trivial usage protocol, the technique does not provide much value.
- As opposed to automated verification tools, EPAs require active engagement from the software developer during the validation tasks. Despite this limitation, in Chapters 6 and 7 we show that users were generally proficient with understanding EPAs.
- Our EPA generation tool currently supports:
  - Pre/postcondition specifications written in the CVC [BB04] language.
  - C source code. Some of the examples analysed throughout this work were originally found in other languages such as C# and Java. In those cases, manual translations were produced and made available to the research community.

1.3. Roadmap

The rest of this document is structured as follows. Part I is comprised of Chapters 1 and 2, where we informally present EPAs and show how they can drive an API validation process (an extended version was first presented in [dCBGU12b]).

Part II is devoted to EPA construction. Chapter 3 introduces the formal underpinnings of our technique, including a formal definition for EPAs (first introduced in [dCBGU09]. In Chapter 4 we offer two EPA construction algorithms (originally presented in [dCBGU12a] and [dCBGU11], respectively). In Chapter 5 we introduce the Contractor tool, discuss its implementation details and analyse how it performs on a series of real-life APIs.

Part III presents the empirical evaluation of our approach. Chapter 6 consists of a series of case studies in which domain experts were asked to use EPAs to validate a number of industrial-strength APIs. Chapter 7 offers a study on EPA expressiveness and understanding, including the report on three controlled user studies.

We close this work in Part IV. We present and discuss related work in Chapter 8 and conclude in Chapter 9 with some final words and an outlook of the road ahead.

1.4. Dissemination of Results

Abbreviated versions of the results presented in this thesis have been originally published by the authors in [dCBGU09, dCBGU12a, dCBGU11, ZBdC+11]. At the time of this writing, the results presented in Chapter 7 have been submitted for consideration to International Conference on Software Engineering (ICSE) 2013.

CHAPTER 2

Motivation

In this chapter we illustrate the difficulties of validating APIs using two toy examples. We first describe how the proposed approach can identify issues in an API specification. We then show how our approach is also valid when dealing with API implementations.

2.1. Validating API Specifications using Behaviour Abstractions

Consider the specification of a circular buffer given in Figure 2.1. The specification includes three state variables: a represents an integer array with slots that the buffer uses for storing data, wp is a pointer to the first available slot for storing new data, and rp is a pointer to the last slot from which data was read. The idea is that wp points to a slot further ahead than the slot pointed to by rp and that the slots in between are those that have been written but not yet read. Of course, the fact that this is a circular buffer makes the notion of "further ahead" slightly more complicated to express formally. The specification includes pre and postconditions for two actions applicable to circular buffers: `read` and `write`. Writing requires the buffer to have empty slots and results in a circular buffer that has incremented by one its writing pointer unless it has reached the array limit, in which case the writing pointer is set to 0. Reading requires the buffer to have slots with unread data and updates its reading pointer using the same strategy as write uses for wp. Finally, the specification includes an invariant which requires the circular buffer to have more than three slots for storing data[1] and requires both pointers to be within the bounds of the circular buffer, i.e., between 0 and $|a| - 1$, and there is a condition over the acceptable starting states for circular buffers.

Given the circular buffer specification, how can we validate if it corresponds to the intended behaviour of a circular buffer? As mentioned above, one strategy would be to write another specification (or use an existing one) and verify the contract specification against it using techniques such as model-checking or theorem proving.

For instance, a reviewer might perform an automated analysis capable of checking if the contract specification satisfies some given properties. Techniques such as model checking [CGP99] and in particular infinite state model checking [Esp97b] can verify

[1] Notice however, that the actual storing capacity is always reduced by two.

CircularBuffer
variable a array of integers
variable wp, rp integer
inv $0 \leqslant rp < |a| \ \wedge \ 0 \leqslant wp < |a| \ \wedge \ |a| > 3$
start $|a| > 3 \ \wedge \ rp = |a| - 1 \ \wedge \ wp = 0$
action `write`(integer n)
pre $(wp < rp - 1) \ \vee \ (wp = |a| - 1 \ \wedge \ rp > 0)$
$\vee \ (wp < |a| - 1 \ \wedge \ rp < wp)$
post $rp' = rp \ \wedge \ (wp < |a| - 1 \ \Rightarrow \ wp' = wp + 1)$
$\wedge \ (wp = |a| - 1 \ \Rightarrow \ wp' = 0)$
$\wedge \ (a' = \text{updateArray}(a, wp, n))$
action integer `read()`
pre $(rp < wp - 1) \ \vee \ (rp = |a| - 1 \ \wedge \ wp > 0)$
$\vee \ (rp < |a| - 1 \ \wedge \ wp < rp)$
post $rv = a[rp'] \ \wedge \ wp' = wp \ \wedge \ a' = a$
$\wedge \ (rp < |a| - 1 \ \Rightarrow \ rp' = rp + 1)$
$\wedge \ (rp = |a| - 1 \ \Rightarrow \ rp' = 0)$

Figure 2.1: Specification of a circular buffer

if the entire state-machine defined by a contract specification, as described above, satisfies a property. Theorem proving can check if a property can be directly inferred from the contract specification. The problem with these strategies, in addition to tractability issues, is coming up with the properties to be checked. Some examples of properties that one would want to check against the circular buffer contract are:

1. Initially, the `read` action is enabled after the first `write` action occurs.
2. Either a `write` or a `read` action can be performed at any given moment.
3. The `read` operation is always enabled after any (positive) number of `write` operations.
4. The `write` operation is always enabled after any (positive) number of `read` operations.

The completeness of the set of properties to be checked against the contract is crucial to this strategy: Have we included all the relevant properties? In addition, it requires specifying the intended behaviour twice: Once in an operational pre/post condition style and the other in a more declarative style.

Such strategies can be effective at finding faults but require another specification (namely, the aforementioned desirable properties) and shift the validation problem, as it is now the alternative specification that must be validated.

A complementary approach we propose is to automatically construct a behaviour model such as the one shown in Figure 2.2. In this model the concrete state space of the circular buffer has been abstracted based the set of operations the concrete states enable, that is, the set of operations for which their preconditions hold. Concrete states of a circular buffer that only allow execution of `write` are represented by the {`write`} abstract state, concrete states that allow `write` and `read` are grouped into the {`write`, `read`} abstract state, and all concrete states that only allow to `read` are abstracted into the {`read`} abstract state. Transitions between abstract states exist only if a transition between concrete states they represent exist. Finally, an abstract state is an initial state (a diamond-shaped node) if it abstracts at least one initial concrete state of the circular buffer.

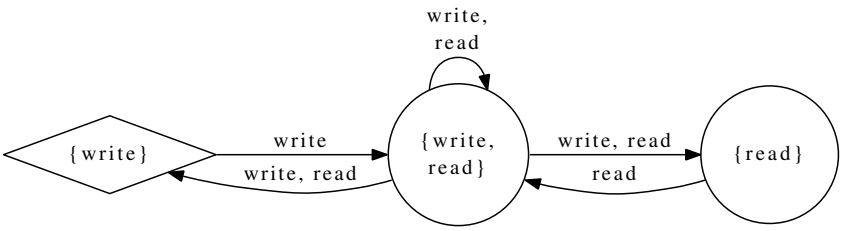

Figure 2.2: Circular buffer finite abstraction

We believe that automated construction of abstractions that consolidate pre/-post condition specifications into one cohesive behaviour model (which quotients states based on the operations they enable) can complement the strategies outlined previously and provide further support for analysis and validation of pre/post specifications. The model of the circular buffer specification shown in Figure 2.2 abstracts away the size of the buffer and brings an infinite state space down to only three abstract states. Furthermore, the three abstract states have a clear and intuitive interpretation in the domain of circular buffers: a circular buffer can be empty ($\{\texttt{write}\}$), full ($\{\texttt{read}\}$), or partially full/empty ($\{\texttt{write}, \texttt{read}\}$).

performed and state 2 allows reading only.

Consider the `write`-labelled transition from state $\{\texttt{write}, \texttt{read}\}$ to $\{\texttt{write}\}$. This transition is suspicious as writing data into a non-empty buffer should not lead to a state that models empty buffers. Similarly, the transition from the state $\{\texttt{write}, \texttt{read}\}$ (non-full) to state $\{\texttt{read}\}$ (full) on label `read` also looks suspicious. These transitions suggest that there could be something in the specification that is not entirely accurate or correct.

To understand why these suspicious transitions appear in the behaviour model it is important to understand the abstraction relation between the model in Figure 2.2 and the specification. The concrete states of the circular buffer are formally abstracted according to following invariants:

- State $\{\texttt{write}\}$: $\texttt{inv} \wedge \texttt{write\_pre} \wedge \neg\texttt{read\_pre}$
- State $\{\texttt{write}, \texttt{read}\}$: $\texttt{inv} \wedge \texttt{write\_pre} \wedge \texttt{read\_pre}$
- State $\{\texttt{read}\}$: $\texttt{inv} \wedge \neg\texttt{write\_pre} \wedge \texttt{read\_pre}$

Let us now try to understand why the transition labelled `write` from states $\{\texttt{write}, \texttt{read}\}$ to $\{\texttt{write}\}$ appears in Figure 2.2 and if this is signalling a problem in the specification. The fact that the transition is enabled in state $\{\texttt{write}, \texttt{read}\}$ follows directly from the choice of level of abstraction of Figure 2.2. State $\{\texttt{write}, \texttt{read}\}$ models all the states of circular buffers in which both `read` and `write` are enabled. So the question to answer is why can `write` lead to state $\{\texttt{write}\}$. The question can be answered by asking how can the invariant of state $\{\texttt{write}\}$ hold if the invariant of state $\{\texttt{write}, \texttt{read}\}$ holds and action `write` occurs; this question can be easily answered automatically with appropriate tool support: If $rp = wp$ holds on top of the invariant for state $\{\texttt{write}, \texttt{read}\}$, then the postcondition for `write` leads to state $\{\texttt{write}\}$.

It turns out that the invariant for circular buffers was missing the condition $rp \neq wp$. The amended specification would yield an abstract behaviour model (see Figure 2.3) without the two suspicious transitions. It is interesting to note the subtlety of this error: The completed invariant is guaranteed to be true by the initial predicate and the postconditions of the two circular buffer actions. Any sequence of

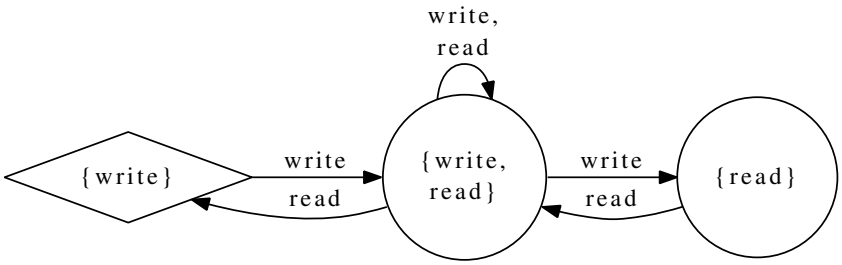

Figure 2.3: Corrected circular buffer finite abstraction

ExtendedCircularBuffer

⋮

inv $0 \leqslant rp < |a| \ \wedge \ 0 \leqslant wp < |a| \ \wedge \ |a| > 3$

⋮

action `reset()`

pre true

post $rp' = wp \ \wedge \ wp' = wp \ \wedge \ a' = a$

Figure 2.4: Circular buffer with `reset`

actions starting from the initial state guarantees $rp \neq wp$ yet the omission becomes a problem if the buffer is extended with legal operations (those that preserve the incomplete invariant) such as the specification shown in Figure 2.4.

In summary, the example above illustrates how an abstract model that integrates the various pieces of information that appear in an API specification supports its validation and aids identifying potential problems it may have. Furthermore, we believe that the specific choice of level of abstraction of the model, the traceability of the abstraction to the specification and to domain-relevant states help identify and fix problems.

2.2. Validating API Implementations using Behaviour Abstractions

Having shown how behaviour abstractions are useful in the context of API specifications, lets now consider API implementations. In other words, we now deal with actual source code that implements a set of operations.

Consider the C source code of Figure 2.5, 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()`).

A similar problem as with the circular buffer arises: 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 against? As mentioned

```
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 int remove(){
39   int ret = l->first->data;
40   node* new_first = l->first->next;
41   free(l->first);
42   l->first = new_first;
43   return ret;
44 }
45
46 int destroy_req() {
47   return l!=NULL;
48 }
49 void destroy() {
50   node* current;
51   node* tmp;
52   current = l->first;
53   l->first = 0;
54   while(current != 0) {
55     tmp = current->next;
56     free(current);
57     current = tmp;
58   }
59   l = 0;
60 }
```

Figure 2.5: A singly-linked list C implementation

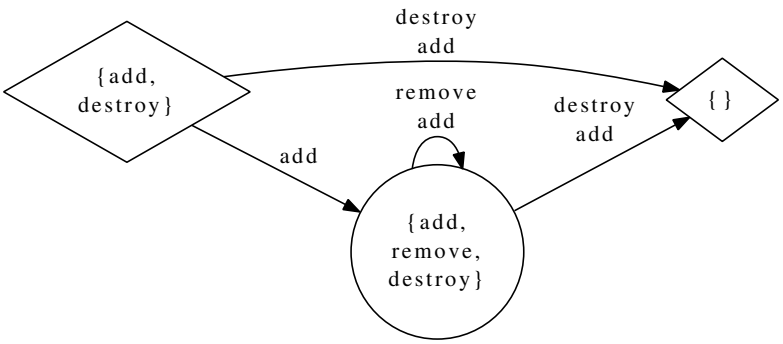

Figure 2.6: Singly-linked list enabledness abstraction

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.

As we did in the case of the circular buffer specification, we propose to automatically extract a behaviour model such as the one shown in Figure 2.6. 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. The {`add`, `destroy`} abstract state groups concrete instances that allow execution of `add` and `destroy`. Abstract state {`add`, `remove`, `destroy`}

groups concrete instances that allow `add`, `remove`, and `destroy`. And the {} abstract state groups all concrete instances that do not allow any operation. Like in the previous example, *initial states* are identified as *diamond-shaped nodes*.

It is simple to see that this model describes states which relate to whether a singly-linked list is empty ({`add`, `destroy`}), non-empty ({`add`, `remove`, `destroy`}), or inactive ({}).

Consider the `remove` operation. It is only featured in a transition that loops over state {`add`, `remove`, `destroy`}. 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 {`add`, `remove`, `destroy`} back to {`add`, `destroy`}, 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 2.5 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 2.6, but with the addition of the missing `remove` transition from {`add`, `remove`, `destroy`} back to {`add`, `destroy`}.

The abstraction in Figure 2.6 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 {}. 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.

2.3. Enabledness Preserving Abstractions

The behaviour models presented so far are called *enabledness-preserving abstractions* (EPAs). They group concrete instances into abstract states according to which actions are *enabled* and which actions are not.

Such behaviour models are permissive. Every legal operation sequence on the original artefact (i.e., the specification or implementation) is included on the EPA'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 original artefact. For instance, the sequence `write` $\rightsquigarrow$ `read` $\rightsquigarrow$ `read` is part of the EPA's language in Figure 2.3 but it is not a legal action sequence according to the circular buffer specification in Figure 2.1, even adding the missing part of the invariant. 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.

According to our previous experience [dCBGU12a], 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 `read`, since a finite model can not keep track of how many times this operation has been invoked with respect to `write` invocations.

The question arises, why choose this particular 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 discuss EPA refinements later in this work and do not discard the possibility that other abstraction levels could also prove useful in helping developers during validation tasks.

In the next two chapters we show how enabledness-preserving abstractions like the one in Figure 2.6 can be built automatically from APIs such as the one depicted in Figure 2.5.

Part II

Defining & Building EPAs

CHAPTER 3

Formal Setting

The purpose of this part of the thesis is to define EPAs and provide EPA construction algorithms, together with their implementation.

EPAs provide an abstract representation of the usage protocol for an API. They do so in a concise manner by grouping those concrete instances of the API that enable the same set of operations.

EPAs are constructed from either the specification or the implementation of an API. Specifications can have different flavours and implementations can be written in a miriad of programming languages. In order to abstract away from this variety, we introduce *action systems*. An action system mainly consists of a series of actions, each having a requires clause and an implementation.

In order to accommodate both API implementations and specifications, action implementations are defined in terms of 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.

3.1. Action Systems

We define an *action system* as the semantic interpretation of the API under analysis. Action systems encode the information known about the input API, whether it is in the form of a specification or a concrete (source code) implementation.

An action system comes with an *initial condition* that indicates which configurations are legal freshly constructed API instances. An action system provides a *system invariant* that characterises the set of legal configurations for the API internal state. It also provides a set of *actions* that constitute the public interface for the API. For each of these actions, it provides a *requires clause* that indicates when is it safe to invoke such action, as well as a *function* that transforms the configuration accordingly when the action is invoked.

Action systems are formally defined as follows.

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

- $init : \mathbb{C} \to \{\text{true}, \text{false}\}$ is the *initial condition*, which indicates if a given configuration is an initial configuration.
- $inv : \mathbb{C} \to \{\text{true}, \text{false}\}$ is the action *system invariant*.

- $Act = \{a_1, \ldots, a_n\}$ is a finite set of *action labels*.
- 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 parameters. Notice that for the sake of simplicity, and without losing generality, we restrict ourselves to consider functions with a single integer parameter.
- F is an Act-indexed set of *functions*. For each action label a, the function $F_a : \mathbb{C} \times \mathbb{Z} \to (\mathbb{C} \cup \bot)$ 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 $\bot$ symbol).

In the rest of this thesis, we assume that the system invariant correctly and precisely characterises legal instances of the action system. In Section 4.4 we discuss the consequences of the violation of this assumption.

We now provide two action system examples, one for the circular buffer specification presented in Figure 2.1 and one for the list implementation given in Figure 2.5.

Example 3.1 (Circular Buffer Action System)**.** *A possible action system for the circular buffer specification given in Figure 2.1 is* $AS = \langle init, inv, Act, R, F \rangle$ *where:*

- *init yields* true *only for configurations that have an array of length at least 3, an rp variable pointing to the last position in the array and a wp variable pointing to 0.*
- *inv returns* true *only if the array is of length at least 3 and both rp and wp are pointing within the array bounds.*
- $Act = \{\texttt{write}, \texttt{read}\}$.
- $R_{\texttt{write}}$ *yields* true *only for configurations on which there is room for an extra element in the array. Informally, this happens when the wp pointer is to "the left" of the rp pointer. Of course, the concept of "left" has to consider the fact that the array is to be interpreted circularly. Formally,* $R_{\texttt{write}}$ *yields* true *when one of the following holds:*

  I. $wp < rp - 1$

  II. $wp = |a| - 1 \ \wedge \ rp > 0$

  III. $wp < |a| - 1 \ \wedge \ rp < wp$

- $R_{\texttt{read}}$ *yields* true *only for configurations on which there is at least one valid item stored in the array. Informally, this happens when the wp pointer is circularly to "the right" of the rp pointer. We omit the formal definition, which is analogous to the previous one.*
- $F_{\texttt{write}}$ *returns a new configuration on which the parameter has been stored at the position pointed by wp. The wp pointer is therefore circularly shifted one position "to the right".*
- $F_{\texttt{read}}$ *returns a new configuration on which the rp pointer is circularly shifted one position "to the right". A distinguished retVal variable in the configuration is created so that it points to the element that was stored in the array at position* $rp + 1$.

Example 3.2 (List Action System)**.** *A possible action system for the list C implementation in Figure 2.5 of the previous section is* $AS = \langle init,\ inv,\ Act,\ R,\ F\rangle$ *where:*

- *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.*
- *inv returns* true *only if* i) *the configuration has a* `NULL` `l` *variable; or* ii) *if* `l` *has a non-negative* `size` *field.*
- $Act = \{\texttt{add}, \texttt{remove}, \texttt{destroy}\}$*. This set of actions indicates the names of the actions exposed in the* public interface *of the list implementation.*
- $R_{\texttt{add}}$ *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* $R_{\texttt{add}}$*.*
- $F = \{F_{\texttt{add}},\ F_{\texttt{remove}},\ F_{\texttt{destroy}}\}$*. Where these functions are the semantic interpretation of the corresponding C functions in Figure 2.5.*

Note that these action systems are rather arbitrary. For instance, we could have decided to exclude the `destroy` operation from the public interface, removing it from the actions set, which would have characterised a system with a smaller state space.

When considering API specifications, the implementation $F_a(c, p)$ for each action a is obtained as the (only) configuration c' that satisfies the postcondition of $a(p)$ invoked on c. If the postcondition for an action is non-deterministic (i.e., there are many possible c' that satisfy such condition), the definition for action systems becomes a little bit more elaborate. For the sake of presentation we restrict ourselves to the case of deterministic pre/postcondition contracts in the rest of this work.

On the other hand, when dealing with API implementations 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 2.5. 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 2.5.

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 3.2 (Action System Semantics)**.** Given an action system $AS = \langle init,\ inv,\ Act,\ R,\ F\rangle$, we say that its *semantics* is provided by an LTS $L = \langle \mathbb{A},\ \mathbb{S},\ \mathbb{S}_0,\ \Delta\rangle$ satisfying $\mathbb{A} = Act \times \mathbb{Z}$, $\mathbb{S} = \{c \in \mathbb{C} \mid inv(c) = \text{true}\}$ and $\mathbb{S}_0 = \{c \in \mathbb{S} \mid init(c) = \text{true}\}$. Also, for each $c \in \mathbb{S}$ and for each $a \in Act$ and $p \in \mathbb{Z}$ such that $R_a(c, p) = \text{true}$, if $F_a(c, p) = c'$ and $inv(c') = \text{true}$ then $\Delta\big(c, (a, p)\big) = c'$. The transition function is not defined for 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 3.3 (List underlying LTS)**.** Given the action system described in Example 3.2, Figure 3.1 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).

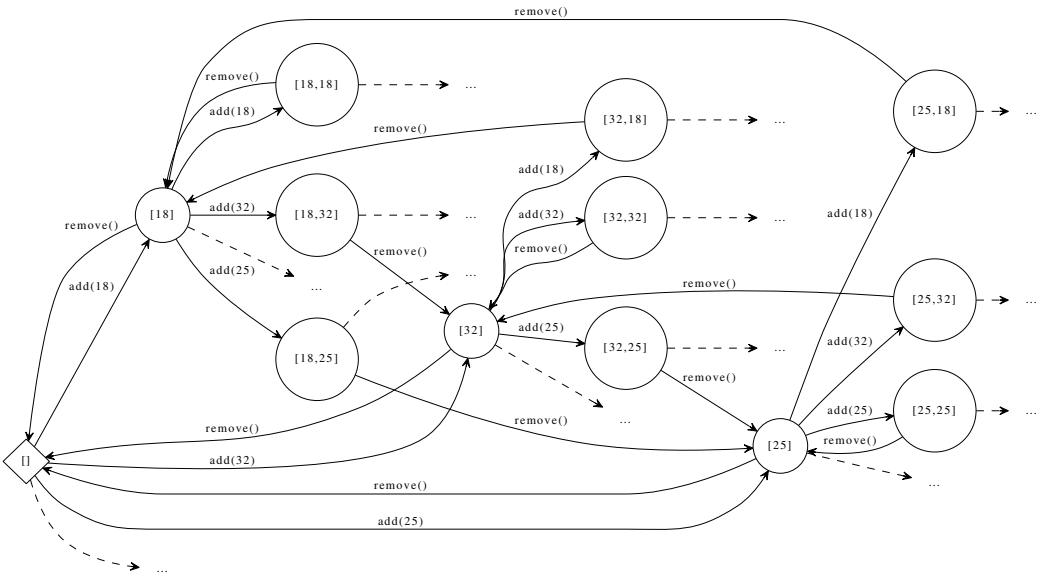

Figure 3.1: Finite fragment of the list underlying LTS

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.

3.2. Enabledness Abstractions

Now that we have defined the state space of an action system by means of its LTS, we need to define a proper level of abstraction in order to obtain a finite representation. Our experience indicated that grouping LTS states for which the same set of actions are enabled is an abstraction level that provides a good compromise between size and precision. In the next chapters we will discuss and provide evidence about this claim. For now, we introduce its formal underpinnings.

First of all, we present the concept of *enabledness equivalence*. Two configurations are enabledness equivalent when they allow the same set of operations to be invoked.

Definition 3.3 (Enabledness Equivalence)**.** Let $AS = \langle init,\ inv,\ Act,\ R,\ F \rangle$ be an action system. Given 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 one step.

Example 3.4 (Enabledness Equivalent List Instances)**.** Continuing with the list example from Figure 2.5, consider the following instances:

l_1 : A list of size 3, with nodes carrying the integers $1, 3, 5$.

l_2 : A list of size 2, with nodes carrying the integers $39, 10$.

l_3 : A list of size 0.

Both l_1 and l_2 enable the same set of actions, namely the `add`, `remove` and `destroy` operations and therefore $l_1 \equiv_e l_2$. On the other hand l_3 only enables `add` and `destroy` so it is not equivalent to any of the two other instances.

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. 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. In other words, we quotient the action system semantics using the enabledness equivalence relation. Furthermore, this abstraction is able to *simulate any path* in the LTS describing the action system semantics.

Definition 3.4 (Enabledness-preserving Abstraction)**.** Given an action system $AS = \langle init,\ inv,\ Act,\ R,\ F \rangle$ and its LTS $L = \langle \mathbb{A},\ \mathbb{S},\ \mathbb{S}_0,\ \Delta \rangle$, we say $M = \langle \Sigma,\ S,\ S_0,\ \delta \rangle$ 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 3.5 (Enabledness-preserving Abstraction for the Circular Buffer)**.** Figure 2.2 presents an enabledness-preserving abstraction for the action system given in Example 3.1.

Example 3.6 (Enabledness-preserving Abstraction for the List)**.** Figure 2.6 depicts an enabledness-preserving abstraction for the action system presented in Example 3.2.

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 is *not* possible from c.

Definition 3.5 (Predicate of an Action Set)**.** Let $AS = \langle init,\ inv,\ Act,\ R,\ F \rangle$ be an action system . The *predicate of a set of actions* $A \subseteq Act$ is the function $\text{pred}_A : \mathbb{C} \to \{\text{true}, \text{false}\}$ defined as:

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

Example 3.7 (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:

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

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

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

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

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 a lemma about α.

Lemma 3.2.

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

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

Back to our proof of Theorem 3.1, 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 3.2, $\mathbb{S} = \{c \in \mathbb{C} \mid inv(c) = \text{true}\}$ 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 Lemma 3.2, 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 3.2, 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 Lemma 3.2, 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. □

3.3. Closing Remarks

In this chapter we formally introduced action systems and EPAs. We also presented Theorem 3.1, which provides a straightforward (yet very inefficient) way of constructing EPAs by enumerating every single possible abstract state and testing for the presence of each possible transition. In the next chapter we present more efficient construction algorithms.

CHAPTER 4

EPA Construction

In this section we present the formal underpinnings behind the construction of the enabledness-preserving abstraction of an action system. We begin by presenting two construction algorithms in Section 4.1 and Section 4.2. In Section 4.3 we describe how the queries performed by these two algorithms are actually solved. The chapter finishes with a discussion about the technique's assumptions in Section 4.4.

4.1. Enumeration Algorithm

A trivial algorithm using the concepts of Theorem 3.1 would require $n \times \Omega(2^{n-1} \times \times 2^n)$ transition tests, where n is the number of actions in the action system. This is because, for each action, it appears in exactly 2^{n-1} abstract states, and it could potentially advance to any abstract state. In a straightforward implementation, space complexity would also be exponential in n since the set of states would have to be kept in memory while computing transitions.

In this section we present a more sophisticated algorithm which splits the construction problem into three parts: *i)* obtaining a set of candidate states, *ii)* computing the transitions between these states, and *iii)* restricting the result to the reachable part.

The first part could be easily accomplished by just enumerating all the possible states, but this would result in a very expensive transition-computation phase. Instead, we construct a set of candidate states by calculating enabledness dependencies among actions, yielding a set of states which is usually much smaller than 2^n but still contains any reachable state in the resulting EPA. The second part takes the set of candidate states and explores every possible transition between them in a standard manner. The complexity of the second phase is heavily dependent on the size of the candidate set constructed in the first phase. Finally, the third phase restricts the result to only those connected subgraphs which contain at least one initial state.

First of all, we define the notion of enabledness dependency between actions. We say that actions a and b are dependent if either: *i)* every time that a is enabled then b is also enabled, *ii)* every time that a is enabled then b is disabled, *iii)* every time that a is disabled then b is enabled, or *iv)* a is disabled implies that b is disabled.

Definition 4.1 (Enabledness Dependencies)**.** *Let* $AS = \langle init,\ inv,\ Act,\ R,\ F \rangle$ *be an action system. We define the following* enabledness dependency relations *in* $Act \times Act$*:*

- $D^{++} \stackrel{def}{=} \{(a,b) \mid \forall c \in \mathbb{C}, p \in \mathbb{Z}.\ inv(c) \wedge R_a(c,p) \Rightarrow R_b(c,p)\}$
- $D^{+-} \stackrel{def}{=} \{(a,b) \mid \forall c \in \mathbb{C}, p \in \mathbb{Z}.\ inv(c) \wedge R_a(c,p) \Rightarrow \neg R_b(c,p)\}$
- $D^{-+} \stackrel{def}{=} \{(a,b) \mid \forall c \in \mathbb{C}, p \in \mathbb{Z}.\ inv(c) \wedge \neg R_a(c,p) \Rightarrow R_b(c,p)\}$
- $D^{--} \stackrel{def}{=} \{(a,b) \mid \forall c \in \mathbb{C}, p \in \mathbb{Z}.\ inv(c) \wedge \neg R_a(c,p) \Rightarrow \neg R_b(c,p)\}$

Example 4.1 (Enabledness Dependencies in the List Action System)**.** *In the context of the* `List` *action system presented in Example 3.2, the enabledness dependency relations are as follows:*

$$
\begin{aligned}
D^{++} &= I \cup \{(\texttt{remove},\texttt{add}),(\texttt{remove},\texttt{destroy}),(\texttt{add},\texttt{destroy}),(\texttt{destroy},\texttt{add})\}\\
D^{+-} &= \varnothing\\
D^{-+} &= \varnothing\\
D^{--} &= \{(b,a) \mid (a,b) \in D^{++}\}
\end{aligned}
$$

Where $I = \{(\texttt{add},\texttt{add}),(\texttt{remove},\texttt{remove}),(\texttt{destroy},\texttt{destroy})\}$ *is the reflexive relation involving the three actions.*

Intuitively, given a set of actions, we will say that it is compliant with the enabledness dependencies relations if it satisfies all the restrictions that they impose.

Definition 4.2 (Enabledness Dependencies Compliance)**.** *Given an action system* $AS = \langle init,\ inv,\ Act,\ R,\ F \rangle$ *and its enabledness dependency relations* $D^{++}, D^{+-}, D^{-+}, D^{--}$*, we say that a set of actions* $A \in 2^{Act}$ complies with the enabledness dependencies *if all the following conditions hold:*

1. *For every* $(a,b) \in D^{++}$*, if* $a \in A$ *then* $b \in A$*.*
2. *For every* $(a,b) \in D^{+-}$*, if* $a \in A$ *then* $b \notin A$*.*
3. *For every* $(a,b) \in D^{-+}$*, if* $a \notin A$ *then* $b \in A$*.*
4. *For every* $(a,b) \in D^{--}$*, if* $a \notin A$ *then* $b \notin A$*.*

Example 4.2 (List Action System Enabledness Dependencies Compliance)**.** *Given the enabledness dependencies for the* `List` *action system presented in Example 4.1:*

- $A = \{\texttt{add}\}$ *is not dependency compliant. In particular, since* $(\texttt{add},\texttt{destroy}) \in D^{++}$ *and* $\texttt{add} \in A$ *then* `destroy` *should also be part of* A*, but it is not.*
- $B = \{\texttt{add},\texttt{remove},\texttt{destroy}\}$ *is dependency compliant.*
- $C = \varnothing$ *is dependency compliant.*

The enabledness dependency relations are straightforwardly computed using the following algorithm.

Definition 4.3 (Enabledness Dependencies Computation Algorithm)**.** *Given an action system* $AS = \langle init,\ inv,\ Act,\ R,\ F \rangle$*, we construct the enabledness dependency relations* D^{++}, D^{+-}, D^{-+} *and* D^{--} *using the following procedure.*

Procedure BUILDDEPENCENCIES

Input: An action system $AS = \langle Act,\ F,\ R,\ inv,\ init \rangle$

Output: The dependency relations $D^{++}, D^{+-}, D^{-+}, D^{--}$ for the given action system.

```
1  D^++ ← ∅;
2  D^+- ← ∅;
3  D^-+ ← ∅;
4  D^-- ← ∅;
5  for a ∈ Act do
6  |  for b ∈ Act do
7  |  |  if ∀c ∈ ℂ, p ∈ ℤ. inv(c) ∧ R_a(c,p) ⇒ R_b(c,p) then
8  |  |  |  D^++ ← D^++ ∪ {(a,b)};
9  |  |  else if ∀c ∈ ℂ, p ∈ ℤ. inv(c) ∧ R_a(c,p) ⇒ ¬R_b(c,p) then
10 |  |  |  D^+- ← D^+- ∪ {(a,b)};
11 |  |  end
12 |  |  if ∀c ∈ ℂ, p ∈ ℤ. inv(c) ∧ ¬R_a(c,p) ⇒ R_b(c,p) then
13 |  |  |  D^-+ ← D^-+ ∪ {(a,b)};
14 |  |  else if ∀c ∈ ℂ, p ∈ ℤ. inv(c) ∧ ¬R_a(c,p) ⇒ ¬R_b(c,p) then
15 |  |  |  D^-- ← D^-- ∪ {(a,b)};
16 |  |  end
17 |  end
18 end
```

To analyse the time complexity of this algorithm, we count the number of logical implications which need to be solved. This number drives the resulting execution time since solving each of these implications is much more expensive than the other operations in the algorithm (initialising sets, adding elements to sets). More concretely, the number of logical implications is bounded by $4 \times n^2$, where n is the amount of actions. Notice that $(a, b) \in D^{--}$ is equivalent to $(b, a) \in D^{++}$, therefore reducing the total number of logical implications that need to be solved.

Furthermore, the algorithm in Definition 4.3 is rather naïve. Our tool, discussed later in Chapter 5, implements a few optimisations. In a first round, only dependencies among labels a_i and a_j with $i \leqslant j$ are calculated. This information is then propagated using a standard fix-point algorithm to calculate the rest of the dependencies. This reduces almost in half the number of logical implications that need to be solved in order to compute the enabledness dependency relations.

We now postulate that enabledness-dependencies are sound, in the sense that abstract states not compliant with them are infeasible.

Lemma 4.1. *Given a state $A \in 2^{Act}$, if $pred_A$ is satisfiable then A is compliant with the enabledness dependency relations.*

Notice that the converse is not true: there exist states that are compliant with the enabledness dependency relations but are not consistent. For instance, consider an action system with an internal state consisting of integer variables x, y, z, true as invariant and actions a_1, a_2, a_3 with requires clauses $x < y$, $y < z$ and $z < x$ respectively. There are no enabledness dependencies, therefore the state $\{a_1, a_2, a_3\}$ is compliant, but its state predicate is not satisfiable.

Once that we have calculated the enabledness dependency relations, we can proceed to enumerate all the states that comply with these restrictions. The following algorithm provides an efficient way to do so.

Definition 4.4 (Enumerating States that Comply with Enabledness Dependencies)**.** *Given an action system $AS = \langle init,\ inv,\ Act,\ R,\ F \rangle$ and its enabledness dependencies relations D^{++}, D^{+-}, D^{-+} and D^{--}, we compute a set of states $S^\star$ given as the result of* ENUM$(\varnothing,\ 1)$.

Procedure ENUM(current, i)

Input: An action system $AS = \langle Act,\ F,\ R,\ inv,\ init \rangle$, its dependency relations $D^{++}, D^{+-}, D^{-+}, D^{--}$ and an index i pointing to the action under consideration in the Act actions array.

Output: A set of states $S^\star$ complying with the given dependencies.

```
1  if i > n then
2  |  if ∃c ∈ ℂ. pred_current(c) then
3  |  |  return {current};
4  |  else
5  |  |  return ∅;
6  |  end
7  else if a_i ∉ current ∧ (¬a_i) ∉ current then
8  |  c_1 ← current ∪ {a_i};
9  |  c_1 ← c_1 ∪ {b | (a_i, b) ∈ D^{++}};
10 |  c_1 ← c_1 ∪ {(¬b) | (a_i, b) ∈ D^{+-}};
11 |  c_2 ← current ∪ {(¬a_i)};
12 |  c_2 ← c_2 ∪ {b | (a_i, b) ∈ D^{-+}};
13 |  c_2 ← c_2 ∪ {(¬b) | (a_i, b) ∈ D^{--}};
14 |  return ENUM(c_1, i + 1) ∪ ENUM(c_2, i + 1);
15 else
16 |  return ENUM(current, i + 1);
17 end
```

The main idea behind this recursive algorithm is to gradually explore all the states, while cutting branches that are known to violate the given dependencies. Initially we start with $current = \varnothing$ and consider the first action a_1. In line 7, since neither this action nor its negation are part of the empty set, we proceed to make 2 recursive invocations:

1. In lines 8–10 we construct the argument c_1 for the first recursive invocation. We include a_1 as part of the new *current* set. Furthermore, when available, we use the information from the enabledness dependencies to extend the *current* set with other actions or their negations.

2. Similarly, in lines 11–13 we construct the argument c_2 for the second recursive information by including $\neg a_1$ and possibly other actions derived from the dependencies.

Finally, in line 14 we make the recursive invocations and incrementing the second parameter so that the algorithm now considers a_2.

In a general recursive step, there are 3 possible scenarios:

- We reach the end $i > n$. In this case *current* already holds a full set of actions or their negations. By construction, *current* satisfies the enabledness dependencies as we never add actions (or their negations) that disrupt them (see lines 8–13). The only remaining thing to do is to check whether it complies with the action system invariant and if so return it as a valid state[1].

- Alternatively, if $i \leqslant n$ and neither a_i nor its negation are part of *current* we are in a situation analogous to the one analyzed above in the context of the first invocation.

[1]Notice that, since states are defined only by the set of actions that they contain, all the negated actions are implicitly dropped.

- Finally, if $i \leqslant n$ and either a_i or its negation are part of *current*, we don't need to make two recursive invocations as the information for this action (or its negation) was previously introduced via dependencies in an invocation $j < i$. It suffices to make a single recursive invocation to consider action a_{i+1}.

Lemma 4.2. *The set of candidate states $S^\star$, as constructed in Definition 4.4, satisfies that it is equal to the set of consistent states that comply with the enabledness dependency relations.*

Once that the set $S^\star$ of candidate states has been constructed, we need to construct the transitions between states in $S^\star$. This is performed using the following algorithm.

ALGORITHM 1: EPA Construction by Enumeration

Input: An action system $AS = \langle Act,\ F,\ R,\ inv,\ init\rangle$, and a set of candidate states $S^\star$
Output: The EPA $M = \langle \Sigma,\ S,\ S_0,\ \delta\rangle$.

```
1  S_0 ← {s ∈ S* | ∃c ∈ ℂ. pred_s(c) ∧ init(c)};
2  Σ ← Act;
3  δ(s, a) ← ∅   ∀s, a;
4  for A, B ∈ S* do
5  |  for each action a ∈ A do
6  |  |  if ∃c. pred_A(c) ∧ ∃p. R_a(c, p) ∧ pred_B(F_a(c, p)) then
7  |  |  |  δ(A, a) ← δ(A, a) ∪ {B};
8  |  |  end
9  |  end
10 end
```

In this algorithm we test each of the candidate states in $S^\star$ to see if they are initial states (which requires $|S^\star|$ queries, or $O(2^n)$). We then initialise the transition function as empty for any input and proceed to check if any pair of states is reachable using enabled transitions in the departing state (which requires $|S^\star|^2 \times |Act|$ queries, or $O(n \times 2^{2n})$).

We can now postulate that the abstraction constructed by Algorithm 1 is indeed an EPA.

Theorem 4.3. *Given an action system $AS = \langle init,\ inv,\ Act,\ R,\ F\rangle$, then M as built by Algorithm 1 using the set $S^\star$ of candidate states as given by Definition 4.4 is an EPA of AS.*

The proof for this theorem is based on the fact that Algorithm 1 performs an exhaustive exploration which complies with Theorem 3.1 over the set of states $S^\star$. All the states which are left out of this exploration (namely, $2^{Act} \setminus S^\star$) would never be part of the final result since their state predicates are unsatisfiable, as implied by Lemmas 4.1 and 4.2.

The final EPA construction phase, which is the restriction of the resulting abstraction to its reachable fragment, is entirely standard and will not be analysed here. Furthermore, notice that this phase could be combined with the transition generation phase following standard BFS or DFS exploration patterns.

We now proceed to analyse the time complexity of the full construction process introduced so far. As mentioned before, the enabledness dependencies calculation needs $O(n^2)$ queries, where n is the number of actions. The construction of the set of candidate states $S^\star$ requires one query for each state that is compliant with the enabledness dependencies. If we have no dependencies at all, then we need $O(2^n)$ queries, which is the worst case. However, in practice the set of compliant states

and consistent states in the resulting abstraction is very similar, as discussed in the following chapters. As mentioned before, the transition calculation phase requires $O(|S^\star|^2 \times n)$ queries. Finally, the restriction to the reachable fragment requires no queries.

4.2. On-the-fly Exploration Algorithm

The enumeration algorithm presented previously in this chapter has the potential drawback that it has to keep in memory the set of abstract states that satisfy the enabledness dependencies. In this section we present a different construction strategy based on an on-the-fly exploration of the reachable abstract states. Instead of computing enabledness dependencies, we iteratively explore the abstract state starting from the initial states. More concretely, Algorithm 2, presented in this section, performs a Breadth-first search (BFS) exploration of the enabledness state space. We thus mitigate the need to exhaustively enumerate all the possible 2^n abstract states for a program with n actions. Using this exploration strategy guarantees that we only consider reachable abstract states, avoiding the need for the last phase of the strategy presented in the previous section (restriction to the reachable fragment).

ALGORITHM 2: EPA Construction

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

$\Sigma = Act$; $S = \varnothing$;
$\delta(A, a) = \varnothing, \quad \forall A, a$;
$A^- = \{a \in Act \mid \forall c.\ init(c) \Rightarrow \neg\exists p.\ R_a(c, p)\}$;
$A^+ = \{a \in Act \mid \forall c.\ init(c) \Rightarrow \exists p.\ R_a(c, p)\}$;
$S_0^C = \{A \subseteq Act \mid A^+ \subseteq A,\ A^- \cap A = \varnothing\}$;
$S_0 = \{A \in S_0^C \mid \exists c.\ \mathrm{pred}_A(c) \wedge init(c)\}$;
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**
    $B^- = \{b \in Act \mid \forall c, p.\ \mathrm{pred}_A(c) \wedge R_a(c, p) \Rightarrow \neg\exists p'.\ R_b(F_a(c, p),\ p')\}$;
    $B^+ = \{b \in Act \mid \forall c, p.\ \mathrm{pred}_A(c) \wedge R_a(c, p) \Rightarrow \exists p'.\ R_b(F_a(c, p),\ p')\}$;
    $S^C = \{B \subseteq Act \mid B^+ \subseteq B,\ B^- \cap B = \varnothing\}$;
    for *each state* $B \in S^C$ **do**
      if $\exists c.\ \mathrm{pred}_A(c) \wedge \exists p.\ R_a(c, p) \wedge \mathrm{pred}_B(F_a(c, p))$ **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 3.1. 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, as we discuss in Section 5.2) 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 3.4.

Theorem 4.4. Given an action system AS, M as built by Algorithm 2 is an EPA of AS.

The proof for this theorem is based on the fact that the abstraction constructed by Algorithm 2 is the reachable fragment of the abstraction presented in Theorem 3.1.

4.3. Solving the Algorithm Queries

Algorithms 1 and 2 are templates that describe how to construct EPAs. In other words, they stipulate *which* queries need to be performed, but not *how* to solve them. In this section we deal with the problem of providing effective answers to these queries.

First of all, 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^- in Algorithm 2, the validity check $\forall c.\ \mathit{init}(c) \Rightarrow \neg\exists p.\ R_a(c,p)$ 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 4.5. Let AS be an action system, and let M be built by either of the following options:

- Algorithm 1 dealing with uncertainty as follows: *a)* If uncertain when deciding enabledness dependencies, do not include the action in the relation. *b)* If uncertain in line 6 then the **then**-branch is executed.
- Algorithm 2 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 3.1: *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 general, in the rest of this work we will still refer to this potentially larger M as EPA. Notationally, we will mark uncertain transitions by suffixing them with a ? symbol.

In the rest of this section, we present two operationalisation strategies for the construction algorithms.

4.3.1. Construction via Satisfiability Queries

We first deal with the case in which the input action system is obtained from a specification (i.e., as opposed to an implementation).

In such a scenario, for each action a, the action system carries a symbolic representation of its function F_a. In other words, we have a postcondition for each action. Therefore, each step of Algorithms 1 and 2 can be encoded as a satisfiability query.

Example 4.3 (Satisfiability Query for the Circular Buffer)**.** *For instance, in order to compute line 13 of the algorithm, given a set of actions A, an action $a \in A$ and another action b we need to decide whether:*

$$\forall c, p.\ pred_A(c) \ \wedge\ R_a(c,p) \ \Rightarrow\ \exists p'.\ R_b\big(F_a(c,p),\ p'\big) \tag{4.1}$$

In the context of the action system in Example 3.1, we instantiate a, b and A as follows:

$$\begin{aligned} a &= \texttt{write} \\ b &= \texttt{read} \\ A &= \{\texttt{write}, \texttt{read}\} \end{aligned}$$

With these values, we now have:

$$\begin{aligned} pred_A(c) = pred_{\{\texttt{write},\texttt{read}\}}(c) &= inv(c) \ \wedge\ \exists p. R_{\texttt{write}}(c,p) \ \wedge\ R_{\texttt{read}}(c) \\ inv(c) &= 0 \leqslant c(rp) < |c(a)| \ \wedge\ 0 \leqslant c(wp) < |c(a)| \ \wedge\ |c(a)| > 3 \\ R_a(c,p) = R_{\texttt{write}}(c,p) &= c(wp) < c(rp) - 1 \ \vee\ (c(wp) = |c(a)| - 1 \ \wedge\ c(rp) > 0) \ \vee \\ &\quad (c(wp) < |c(a)| - 1 \ \wedge\ c(rp) < c(wp)) \\ R_b(c,p) = R_{\texttt{read}}(c,p) &= c(rp) < c(wp) - 1 \ \vee\ (c(rp) = |c(a)| - 1 \ \wedge\ c(wp) > 0) \ \vee \\ &\quad (c(rp) < |c(a)| - 1 \ \wedge\ c(wp) < c(rp)) \end{aligned}$$

$$F_a(c,p) \;=\; F_{\texttt{write}}(c,p) \;=\; c' \text{ such that } \begin{array}{l} c'(rp) = c(rp) \;\wedge \\ (c(wp) < |c(a)| - 1 \;\Rightarrow\; c'(wp) = c(wp) + 1)) \;\wedge \\ (c(wp) = |c(a)| - 1 \;\Rightarrow\; c'(wp) = 0) \;\wedge \\ c'(a) = \text{updateArray}(c(a), c(wp), c(n)) \end{array}$$

Replacing these values in formula 4.1 yields a rather lengthy (yet shallow) expression.

The rest of the algorithm steps translate similarly to satisfiability queries. These queries can then be fed to a satisfiability modulo theories (SMT) solver such as CVC3 [BB04], Yices [DdM06] or Z3 [DMB08]. Naturally, since SMT solving is in general undecidable, we must accommodate uncertain responses coming from these tools. In those scenarios, Theorem 4.5 provides a framework that allows us to obtain an overapproximated (yet still valid) abstraction.

4.3.2. Construction via Code Reachability Queries

In the previous section we dealt with the case in which the input action system has a symbolic representation of the functions governing each action. Since we also want to obtain EPAs from source code, in principle we do not have such a symbolic representation and therefore, unlike the previous case, we can not use a theorem prover in this context. In this section we explain how we can fulfill the tasks prescribed by each step of Algorithm 2 by resorting to code reachability queries[2].

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

$$\forall x.\ \varphi(x) \Rightarrow \psi\big(F(x)\big) \tag{4.2}$$

The general strategy to encode will be as follows:

```
procedure GENERAL-QUERY(x)
    if CodeOf[φ](x) = true then
        y ← CodeOf[F](x)
        if CodeOf[ψ](y) = false then
            TARGET
        end if
    end if
end procedure
```

Software model checkers (e.g., BLAST [BHJM07]) are then used to decide whether 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 (4.2) holds.

In some cases the formula to analyse involves extra parameters, 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.

Query for line 3

For instance, given an action a, the step of line 3 of Algorithm 2 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 : ℂ, p : ℤ)
```

[2]Constructing reachability queries to solve the steps of 1 is an analogous task and is left out of the presentation.

```
    if CodeOf[init](c) = true then
        if CodeOf[R_a](c, p) = true then
            TARGET
        end if
    end if
end procedure
```

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 4.4 (Reachability Query for the List)**.** *Continuing with the list example presented in Figure 2.5, 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
        end if
    end if
end procedure
```

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.*

Query for line 4

The rest of the algorithm is required to decide the validity of similar predicates, however not every predicate can be solved using a code reachability query, since reachability can only encode safety properties. For instance, in line 4 of Algorithm 2, 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. We further discuss requires clause splitting in the next section, as well as in Chapter 5.

Moreover, if non-trivial candidate approximations are provided they can be verified correct. Checking the overapproximation 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 overapproximation; 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
        end if
    end if
end procedure
```

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[3]. 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
    end if
end procedure
```

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 we assume $\widehat{R_a}$ 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

[3]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 [Hod97] for a formal definition.

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
    end if
end procedure
```

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 $\forall c.\ \mathit{init}(c) \Rightarrow \widehat{SR_a}(c)$, 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
        end if
    end if
end procedure
```

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.

Query for line 6

To decide the validity of the predicates in the rest of the algorithm, given an action set $A \subseteq \mathit{Act}$ and a configuration c, we need to be able to determine whether $\mathrm{pred}_A(c)$ holds. As requires clauses can be weakened and strengthened, we can calculate a weaker version:

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

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 : ℂ)
    ret ← inv(c)
    for a ∈ A do
        if CodeOf[S̃R_a](c) = false then
            ret ← false
        end if
    end for
    for a ∉ A do
        if CodeOf[ŜR_a](c) = true then
            ret ← false
        end if
    end for
    return ret
```

```
end procedure
```

Using this action set predicate overapproximation we can decide the validity of the predicate in line 6 of Algorithm 2 as a code reachability query as follows:

```
procedure A-IS-INITIAL-STATE(c : ℂ)
    if OVER-PRED-OF-A(c) = true then
        if CodeOf[init](c) = true then
            TARGET
        end if
    end if
end procedure
```

If the TARGET statement is reachable then we add A to the set S_0 of initial states. In ordet to comply with Theorem 4.5, if we are uncertain whether it is reachable or not, we still add the action set as initial state in the abstraction.

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 of Algorithm 2. Namely, given labels $a, b \in Act$ and an action set A, we need to decide if:

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

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:

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

And a logically stronger right-hand side[4]:

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

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

```
procedure b-DISABLED-AFTER-a-FROM-A(c : ℂ, p : ℤ, p' : ℤ)
    if OVER-PRED-OF-A(c) = true then
        if CodeOf[PR_a](p) = true then
            c' ← CodeOf[F_a](c, p)
            if CodeOf[R̃_b](c', p') = true then
                TARGET
            end if
        end if
    end if
end procedure
```

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.

[4]Notice that we use the weaker requires clause $\widetilde{R}_b$, but its negation produces a strengthening effect.

On the other hand, if TARGET is reachable, then b is not necessarily enabled after the execution of a, due to the overapproximations 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.

Query for line 13

Similarly, in line 13 of Algorithm 2 we need to decide the validity of:

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

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

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

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 : 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 CodeOf[SR̂_b](c') = false then
                TARGET
            end if
        end if
    end if
end procedure
```

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^+.

Query for line 16

Now we focus on the validity check in line 16 of Algorithm 2:

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

In order to comply with Theorem 4.5, 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:

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

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
            end if
        end if
    end if
end procedure
```

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 4.5.

Example Run of Algorithm 2

In the rest of this section, we present a step-by-step execution of Algorithm 2 using reachability queries. 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
        end if
    end if
end procedure
```

} 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
        end if
    end if
end procedure
```

} 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
        end if
    end if
end procedure
```

} 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
        end if
    end if
end procedure
```

} 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 ∨ (l.size = 0 ∧ l.first = NULL)
then
        if ¬(l ≠ NULL) then
            TARGET
        end if
    end if
end procedure
```

} 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.

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

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

- $A = \varnothing$

```
procedure OVER-PRED-OF-∅(l : List)
    ret ← l = NULL ∨ l.size ⩾ 0                  // list invariant
    ret ← ret ∧ ¬(l ≠ NULL)                       // add is not enabled
    ret ← ret ∧ ¬(l ≠ NULL ∧ l.size > 0)          // remove is not enabled
    ret ← ret ∧ ¬(l ≠ NULL)                       // destroy is not enabled
    return ret
end procedure

procedure ∅-IS-INITIAL-STATE(l : List)
    if OVER-PRED-OF-∅(l) then
        if l = NULL ∨ (l.size = 0 ∧ l.first = NULL)
then
            TARGET
        end if
    end if
end procedure
```

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

- $A = \{\texttt{add}\}$

```
procedure OVER-PRED-OF-{add}(l : List)
    ret ← l = NULL ∨ l.size ⩾ 0                          // list invariant
    ret ← ret ∧ l ≠ NULL                                   // add is enabled
    ret ← ret ∧ ¬(l ≠ NULL ∧ l.size > 0)                 // remove is not enabled
    ret ← ret ∧ ¬(l ≠ NULL)                                // destroy is not enabled
    return ret
end procedure

procedure {add}-IS-INITIAL-STATE(l : List)
    if OVER-PRED-OF-{add}(l) then
        if l = NULL ∨ (l.size = 0 ∧ l.first = NULL)
then                                                       } TARGET is unreachable. {add} ∉ S0
            TARGET
        end if
    end if
end procedure
```

- $A = \{\texttt{destroy}\}$

```
procedure OVER-PRED-OF-{destroy}(l : List)
    ret ← l = NULL ∨ l.size ⩾ 0                          // list invariant
    ret ← ret ∧ ¬(l ≠ NULL)                                // add is not enabled
    ret ← ret ∧ ¬(l ≠ NULL ∧ l.size > 0)                 // remove is not enabled
    ret ← ret ∧ l ≠ NULL                                   // destroy is enabled
    return ret
end procedure

procedure {destroy}-IS-INITIAL-STATE(l : List)
    if OVER-PRED-OF-{destroy}(l) then
        if l = NULL ∨ (l.size = 0 ∧ l.first = NULL)
then                                                       } TARGET is unreachable. {destroy} ∉ S0
            TARGET
        end if
    end if
end procedure
```

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

```
procedure OVER-PRED-OF-{add, destroy}(l : List)
    ret ← l = NULL ∨ l.size ⩾ 0                          // list invariant
    ret ← ret ∧ l ≠ NULL                                   // add is enabled
    ret ← ret ∧ ¬(l ≠ NULL ∧ l.size > 0)                 // remove is not enabled
    ret ← ret ∧ l ≠ NULL                                   // destroy is enabled
    return ret
end procedure

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 is reachable. {add, destroy} ∈
            TARGET                                          S0
        end if
    end if
end procedure
```

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

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.

  - $a = \texttt{add}$

    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-$\{\texttt{add}, \texttt{destroy}\}(l$: `List`, e : `int`, e' : `int`)
      if Over-Pred-Of-$\{\texttt{add}, \texttt{destroy}\}(l)$ **then**
      $l' \leftarrow \texttt{add}(l, e)$
      if $l' \neq$ `NULL` **then**
      Target
      end if
      end if
      end procedure

      } Target is reachable. `add` $\notin B^-$

    - `remove`

      procedure `remove`-DISABLED-AFTER-`add`-FROM-$\{\texttt{add}, \texttt{destroy}\}(l$: `List`, e : `int`)
      if Over-Pred-Of-$\{\texttt{add}, \texttt{destroy}\}(l)$ **then**
      $l' \leftarrow \texttt{add}(l, e)$
      if $l' \neq$ `NULL` $\wedge\ l'.size > 0$ **then**
      Target
      end if
      end if
      end procedure

      } Target is reachable. `remove` $\notin B^-$

    - `destroy`

      procedure `destroy`-DISABLED-AFTER-`add`-FROM-$\{\texttt{add}, \texttt{destroy}\}(l$: `List`, e : `int`)
      if Over-Pred-Of-$\{\texttt{add}, \texttt{destroy}\}(l)$ **then**
      $l' \leftarrow \texttt{add}(l, e)$
      if $l' \neq$ `NULL` **then**
      Target
      end if
      end if
      end procedure

      } 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-$\{\texttt{add}, \texttt{destroy}\}(l$: `List`, e : `int`)
      if Over-Pred-Of-$\{\texttt{add}, \texttt{destroy}\}(l)$ **then**
      $l' \leftarrow \texttt{add}(l, e)$
      if $\neg(l' \neq$ `NULL`$)$ **then**
      Target
      end if
      end if
      end procedure

      } Target is reachable. `add` $\notin B^+$

    - `remove`

      procedure `remove`-ENABLED-AFTER-`add`-FROM-$\{\texttt{add}, \texttt{destroy}\}(l$: `List`, e : `int`)
      if Over-Pred-Of-$\{\texttt{add}, \texttt{destroy}\}(l)$ **then**
      $l' \leftarrow \texttt{add}(l, e)$
      if $\neg(l' \neq$ `NULL` $\wedge\ l'.size > 0)$ **then**
      Target
      end if
      end if
      end procedure

      } 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' ← add(l, e)                          } TARGET is reachable.
        if l' ≠ NULL then                       } destroy ∉ B+
            TARGET
        end if
    end if
end procedure
```

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

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-∅-USING-add(l : List, e : int)
    if OVER-PRED-OF-{add, destroy}(l) then
        l' ← add(l, e)                          } TARGET is reachable.
        if OVER-PRED-OF-∅(l') then              } ∅ ∈ δ({add, destroy}, add)
            TARGET
        end if
    end if
end procedure
```

  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' ← add(l, e)                          } TARGET is unreachable.
        if OVER-PRED-OF-{add}(l') then          } {add} ∉ δ({add, destroy}, add)
            TARGET
        end if
    end if
end procedure
```

- $B = \{\texttt{remove}\}$

```
procedure {add, destroy}-TO-{remove}-USING-add(l : List, e : int)
    if OVER-PRED-OF-{add, destroy}(l) then
        l' ← add(l, e)                          } TARGET is unreachable.
        if OVER-PRED-OF-{remove}(l') then       } {remove} ∉ δ({add, destroy}, add)
            TARGET
        end if
    end if
end procedure
```

$\vdots$

- $B = \{\texttt{add}, \texttt{remove}, \texttt{destroy}\}$

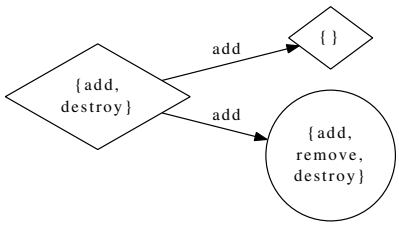

Figure 4.1: Partially explored `List` EPA

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
    end if
  end if
end procedure

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

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

At the end of this step, the algorithm has already explored the `add` action from the {`add`, `destroy`} state. The partially explored EPA is depicted in Figure 4.1.

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

4.4. About the Technique's Assumptions

In the previous sections we presented results that rely on three assumptions. We first assume that the class is equipped with an accurate invariant and requires clauses. We then assume that invariants are accurate. Finally, we assume that requires clauses can be split. In this section we discuss the impact of violating these assumptions. We also discuss about requires clauses correctness.

4.4.1. Dealing with Unannotated Classes

The annotation burden is often presented as one of the biggest obstacles in the adoption of software engineering techniques. In our setting, we need software artefacts to be equipped with requires clauses as well as invariants.

That said, we envision that these annotations required by our technique can be automatically inferred. One possibility is to use invariant mining techniques such as DAIKON [EPG+07]. Another option, in the case of requires clauses, would be to syntactically identify the fragment of the source code that deals with exception checking and heuristically obtain candidate annotations.

In Chapter 6 we analyse a number of case studies over industrial strength API implementations that do not explicitly mark requires clauses or invariants.

4.4.2. 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 difference 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 whose requires clauses are satisfied.

For such cases, we provide an experimental feature that checks the validity of the user-provided invariant on each transition. Extra reachability queries similar to the ones presented in Section 4.3.2 are used for 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.

4.4.3. 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 positive 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.

- $P_1(x_1, \ldots, x_k) \wedge P_2(y_1, \ldots, y_m) \wedge x_i$ 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 4.3.2 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:

$$R_{\texttt{process}}(k) = k \in W$$

Can be rewritten as:

$$\begin{aligned} \widehat{SR}_{\texttt{process}}(W) &= \forall k.\ k \in W \\ \widetilde{SR}_{\texttt{process}}(W) &= \exists k.\ k \in W \end{aligned}$$

With $PR_{\texttt{process}}(k) = \text{true}$.

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 situation in practice.

Nevertheless, we understand this is a limitation in our approach and envision that we need to support a wider variety of patterns in order for our tool to be widely usable.

4.4.4. About Requires Clauses Correctness

Our technique does not impose 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 with weakest preconditions [Dij75]. We analyse what happens when an action a has a requires clause R_a that does not imply the weakest precondition of F_a.

- F_a **could not terminate**. Our reachability queries are built in such a way that the TARGET statement is always after the call to F_a. Therefore, if F_a does not terminate, those traces will not be eligible to be used as reachability witnesses.

- F_a **could terminate, leaving 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.

CHAPTER 5

Implementation

In this chapter we discuss the implementation of our EPA construction algorithms. We begin by introducing CONTRACTOR, an open-source tool developed as part of this work. We then evaluate its scalability in a number of real life API specifications and implementations. Finally, we introduce a series of CONTRACTOR features that were specially designed to assist developers in validation tasks involving EPAs.

5.1. The Contractor Tool

We implemented Algorithms 1 and 2 as a practical tool named CONTRACTOR. CONTRACTOR is open-source and available under a GNU GLP v3 license [GPL07]. The tool, together with all of the case studies discussed in this work is available for download at:

http://lafhis.dc.uba.ar/contractor

In the rest of this section we present CONTRACTOR's architecture and the details of how the satisfiability and reachability queries presented in Section 4.3 are effectively and automatically solved by the tool.

5.1.1. Implementation Notes

From a black-box perspective CONTRACTOR takes an action system description in XML format as input and produces an EPA as output. As we discussed so far, our approach is valid both when the action system describes an API implementation or an API specification. Furthermore, APIs can be implemented in various programming languages, and API specifications have many flavours: first-order logic pre/postcondition specifications, Event-B [MAV05] or Z [Spi92], to name a few. The input XML file for a pre/postcondition API specification simply defines each action as a precondition and a postcondition, as well as giving an invariant and initial condition. On the other hand, the XML file for a C API implementation points to a set of C files; for each action, it provides the name of the C function that implements its requires clause, and also the name of the C function that implements its behaviour.

The resulting EPA can also take several forms: from graphical representations like the ones presented in this work, to machine readable formats that can be fed to other tools and processes.

In order to accommodate the variety of input and output formats we designed CONTRACTOR in a highly modular fashion. Consequently, CONTRACTOR implements both Algorithm 1 and Algorithm 2 but also permits the addition of other construction algorithms without disrupting the rest of the components.

From an object-oriented design patterns perspective [GJHV95], CONTRACTOR implements the construction algorithms as *templates*. They stipulate what *abstract tasks* to perform and in what order, but not how they are implemented. Example abstract tasks are deciding if a pair (a, b) of actions belongs to the enabledness dependency relation D^{++}, deciding if an action b is necessarily enabled after executing a from a particular abstract state, etc. This modular design allows us to have different algorithms and still reuse fragments that they might have in common.

Each input format (e.g., pre/postcondition API specifications, C API implementations, etc.) knows how to solve these abstract tasks. Therefore, each input format implements a *strategy* pattern.

Furthermore, using abstract tasks as the unit of work enabled us to implement a multi-threaded solution. A single (synchronized) queue stores the tasks that need to be performed. A number of worker threads take elements from this queue and notify the queue manager upon completion. Notice that executing one task may produce new tasks that need to be added to the queue. Such is the case, for example, when a new abstract state is visited for the first time and prompts a series of task to be enqueued in order to keep exploring from there.

Input and output are handled via *factories* that understand how to parse API specifications or implementations in the supported formats, as well as producing the desired output file (e.g., XML or PDF).

Overall, CONTRACTOR is 3.4 KLOC long, spread over more than 80 Python modules.

5.1.2. Algorithms Implementation

As we mentioned before, CONTRACTOR implements both Algorithm 1 and Algorithm 2. In this section we discuss the implementation details and some variations. The performance of these implementations is discussed later in Section 5.2.

Construction by Enumeration

When implementing Algorithm 1, we considered two variations.

- The ENUM^D implementation uses the enabledness dependencies information to prune the set of candidate states, as described in Section 4.1.
- The ENUM implementation does not compute the enabledness dependencies and considers every possible abstract state.

The motivation for having an *unoptimised* version is twofold. First, we wanted to see the effect of using enabledness dependencies with respect to a baseline implementation of the enumeration algorithm. Second, in small examples, computing the dependencies is a relatively big task in comparison with the actual states enumeration and its benefits might not pay off.

Construction by Exploration

For Algorithm 2 we considered three variations:

- The EXPL$^{+/-}$ implementation uses the A^+/A^- sets in order to prune the possible destination states as described in Section 4.2.

- The EXPL^D incorporates the enabledness dependencies information when computing the A^+/A^- sets. For instance, if the implementation computes that $a \in A^+$ and it also knows that $(a, b) \in D^{++}$ then it discovers that $b \in A^+$ too, without a need to query the underlying decision engine.

- The EXPL implementation does not compute the A^+/A^- sets. It works as if A^+ and A^- are always empty, in other words, it considers every possible state when exploring.

Finally, the three versions use a caching mechanism to avoid repeatedly asking the same query to the underlying engine. In particular, we implemented the following caches, which are always consulted before querying the engine:

Consistent states: stores all the visited abstract states B such that pred_B in satisfiable.

Inconsistent states: stores all the visited abstract states B such that pred_B is unsatisfiable.

5.1.3. Solving Satisfiability Queries

As we discussed in Section 4.3.1, some action systems have a declarative symbolic representation of the semantics for each action. In such cases, we can solve the algorithms queries via satisfiability checks in a SMT solver.

Currently, CONTRACTOR natively supports CVC3 [BB04] and Yices [DdM06]. The SMT-LIB [RT06] standard language is also supported, so plugging other provers (such as Z3 [DMB08]) should not be a problem.

Supporting several SMT solvers is important since each prover has its strong points. Ideally, we would like to automatically combine, for each query, the answers from different SMT solvers in order to minimise the possibility of having unknown responses. This is currently work in progress.

5.1.4. Solving Reachability Queries

SMT solvers are powerful and efficient tools. However, they require a declarative postcondition for each of the actions in the input action system. In many cases this is an onerous requirement that can not be met. When dealing with API implementations, such as the `Signature`, `Socket` or `SMTPProtocol` classes in the previous section, we need an alternative approach that can work directly with their source code.

For such cases, Section 4.3.2 introduces an operationalisation of the algorithms based on source code reachability queries. In its current version, CONTRACTOR uses the BLAST software model checker to solve these queries.

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 4.3.2, 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 4.5 guarantees that our result is a safe overapproximation.

The exploration for concrete parameter values is trickier 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.

The fact that BLAST only deals with C code, forced us to analyse programs that are written in that programming language. When dealing with programs written in other languages (e.g. the Java `Socket` implementation) we manually translated them to C.

We envision that other backends can be added to CONTRACTOR. For instance, instead of using a software model checker, we could have used a verification-based approach (e.g., [CK05]). We explored such possibility in [ZBdC+11], but this is out of the scope of this thesis.

Another option would be to use a symbolic execution engine (e.g., [KPV03]). 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 [PE07]) for a limited 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 making the user validation process more difficult.

5.2. Quantitative Analysis

In this section we present and analyse how CONTRACTOR performs when constructing the EPAs details on a series of industrial-strength APIs.

More concretely, we want to answer the following research question:

> **R.Q. 1:** Are the EPA construction algorithms efficient enough to deal with industrial-strength APIs and provide an answer in a reasonable amount of time?

We first present the subject APIs we used to answer this research question, and then elaborate on the results.

5.2.1. Subjects

The APIs to which we applied our implementation can be divided in two groups: API specifications and API implementations.

Name	# Actions	Predicate complexity (number of boolean operators) Preconditions	Postconditions	Invariant
`WebFetcher`	4	0	8	4
`ATM`	8	8	16	0
`MS-NSS`	13	23	30	14
`MS-WINSRA`	33	120	241	10

Table 5.1: Case studies specification APIs' size information

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
`MS-PCCRR`	12	251	39	6
`SMTPServer`	9	85	19	4
`SMTPProtocol`	16	510	34	1

Table 5.2: Case studies implementation APIs' size information

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
`MS-PCCRR`	12	9	3
`SMTPServer`	9	9	0
`SMTPProtocol`	16	16	0

Table 5.3: Case studies subjects' requires clauses splitting information

In the first group we have a `WebFetcher` typestate specification from [DF04], `ATM` pre/postcondition specification from [WSCF00] and two Microsoft protocol specifications: .NET `NegotiateStream` (MS-NSS) [MS-08] and `WINS` Replication and Autodiscovery Protocol (MS-WINSRA) [MS-09b].

In the second group we have the `PipedOutputStream`, `Signature`, `ListItr` and `Socket` implementations 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 `MS-PCCRR` class taken from a C# SpecExplorer protocol model [MS-09a].

In Tables 5.1 and 5.2 we provide additional information regarding the size of the subject API specifications and implementations, respectively.

Subject APIs were included according to the following criteria: *i)* APIs that feature rich restrictions in the order in which the actions must be called; *ii)* APIs for which either behaviour documentation or manually-generated behaviour models can be found; *iii)* APIs that have already been analysed using techniques comparable to ours (e.g., [AČMN05, HJM05]); and *iv)* APIs that are of industrial relevance.

When dealing with API implementations we used the existing run-time checks from each class's source code as requires clauses. Table 5.3 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 `MS-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 be used as an exact approximation of the original requires clause from an enabledness point of

Input	Executed queries (running time)				
#Actions, Name	ENUM^D	ENUM	EXPL$^{+/-}$	EXPL^D	EXPL
4, `WebFetcher`	35 (<1s)	**31 (<1s)**	42 (<1s)	60 (<1s)	33 (<1s)
8, `ATM`	**396** (2s)	535 (1s)	438 **(1s)**	504 (2s)	597 (2s)
13, `MS-NSS`	580 (2s)	8454 (23s)	**442 (1s)**	715 (2s)	8507 (39s)
33, `MS-WINSRA` (1st)	**57300 (5m20s)**	n/a (>12h)	102632 (7m38s)	103589 (7m36s)	n/a (> 12h)
33, `MS-WINSRA` (2nd)	**49043** (4m32s)	n/a (>12h)	56993 **(3m58s)**	57764 (4m01s)	n/a (>12h)
33, `MS-WINSRA` (3rd)	**7226 (43s)**	n/a (>12h)	12628 (1m18s)	14517 (1m07s)	n/a (>12h)
4, `PipedOutputStream`	57 (17s)	**56 (10s)**	85 (13s)	97 (14s)	67 (11s)
5, `Signature`	84 **(10s)**	**81** (11s)	127 (15s)	157 (16s)	91 (12s)
5, `ListItr`	266 (11m17s)	**251 (11m11s)**	278 (11m36s)	300 (11m23s)	289 (11m59s)
8, `Socket` (1st)	1992 (8h30m48s)	2012 (8h32m26s)	**401** (1h25m43s)	427 **(1h24m16s)**	1057 (3h03m14s)
8, `Socket` (2nd)	1980 (8h05m55s)	2000 (8h07m38s)	**255** (40m23s)	297 **(39m40s)**	850 (1h43m03s)
12, `MS-PCCRR` (1st)	905 (19m38s)	4723 (28m09s)	**558** (19m44s)	776 **(19m33s)**	4867 (35m13s)
12, `MS-PCCRR` (2nd)	504 **(9m11s)**	4322 (17m49s)	**253** (10m13s)	505 (10m21s)	4337 (21m18s)
9, `SMTPServer`	**273** (42m29s)	434 **(42m12s)**	461 (47m07s)	470 (46m21s)	482 (45m53s)
16, `SMTPProtocol`	**395 (4m51s)**	64644 (53m47s)	979 (10m26s)	882 (8m37s)	65650 (53m48s)

Table 5.4: Executed queries and running times by each implementation

view.

With respect to invariants, we first tried to use a pre-existing one coming from the literature. This is the case in subjects taken from other research papers, as the `ATM` subject.

When no invariant was readily available in the literature, we proceeded by using true as invariant. This is the case in 5 of our subjects.

Finally, when a true invariant was not enough, we (or an expert reviewer, usually a colleague) manually generated an invariant based on exploratory execution and code inspection. This occured in 4 of our subjects.

The invariants for each specific subject are discussed in Sections 6.2 and 6.3.

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 our approach.

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.

5.2.2. Results

The reported running times were taken from an Intel Core i7 (hyper-threaded quad-core) computer with 8 GB of RAM. CONTRACTOR was executed using 8 worker threads running in parallel. The raw data used throughout this section is publicly available in an online spreadsheet located at `http://goo.gl/QxCol`.

Table 5.4 presents the number of executed queries and the running time for each of the case studies. It is divided in two sections: API specifications in the upper half and API implementations in the bottom half. We considered the five algorithm implementations described in Section 5.1.2 and imposed a timeout of 12 hours. The implementation with the least number of queries is highlighted on each row. Similarly, the fastest running implementation is also highlighted. Notice that the fastest implementation is not always the one with the least queries.

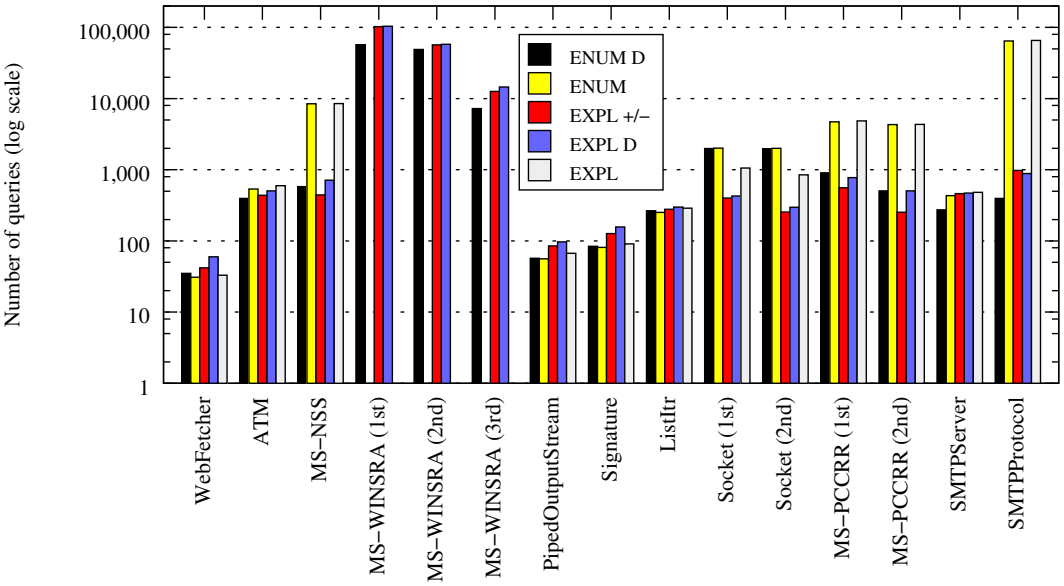

Figure 5.1: Executed queries on each case study

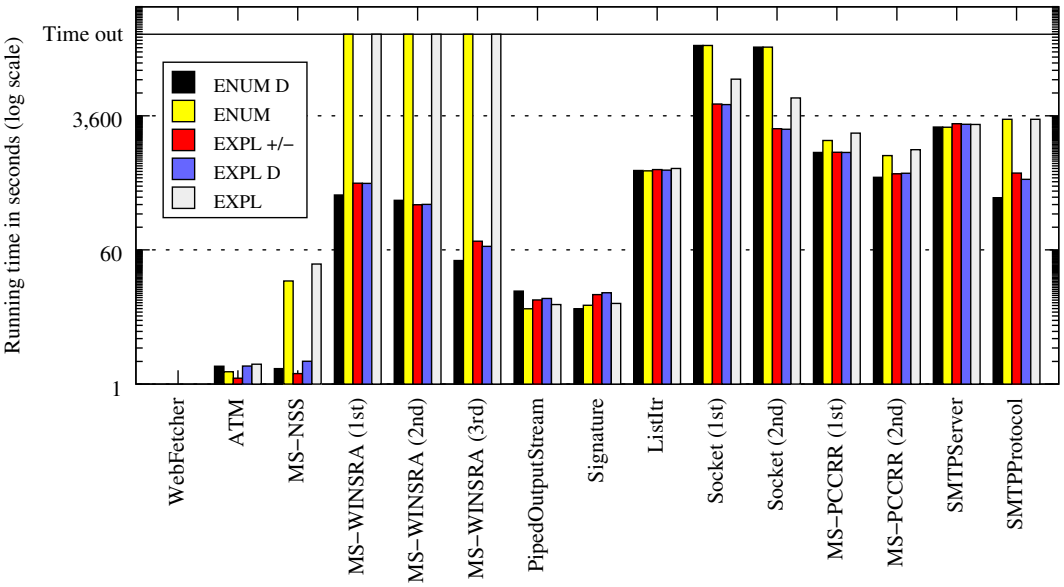

Figure 5.2: Running time on each case study

Figures 5.1 and 5.2 present a graphical representation of the data in Table 5.4.

We now analyse each of the case studies in detail, starting with the computationally expensive ones.

In `MS-NSS` the most effective implementation is Expl$^{+/-}$. It is closely followed by both Enum^{D} and Expl^{D}, but these are slower since they compute the enabledness dependencies. In this particular example, the benefits of having computed these dependencies do not pay off soon enough, making these two implementations slower. The unoptimised versions Enum and Expl are significantly slower as they consider roughly 8000 abstract states, when the reachable fragment in the resulting EPA has only 10 states.

With 33 actions, the `MS-WINSRA` case study is the largest in terms of abstract state space. The unoptimised algorithms fail to explore all the 2^{33} possible action combinations. The fastest implementation in this case is almost always Enum^{D}.

This is due to the fact that this example has rich enabledness dependency sets. For instance, the first version has 656 out of 1056 possible dependencies. Only 105 abstract states comply with these dependencies. On the other hand, the EXPL$^{+/-}$ and EXPL^{D} implementations consider roughly 90,000 states.

In the `ListItr` case study all the implementations perform similarly in terms of time and number of queries. This can be explained by the fact that there are almost no enabledness dependencies in this example; similarly, there are almost no actions in the A^+/A^- sets. The `SMTPServer` case study presents a very similar situation.

The `Socket` case study is dominated by the exploration implementations. There are very few enabledness dependencies (only 22) and rougly 165 actions are included in A^+/A^- sets in the course of the execution. The ENUM^{D} implementation is therefore forced to consider 36 abstract enabledness compliant states, when only 9 are eventually reachable. A similar situation occurs in the `MS-PCCRR` case study.

In the `SMTPProtocol` case study, the resulting EPA only features 2 abstract states (out of 2^{16} possible action combinations). In this context, ENUM^{D} quickly discovers these 2 states, only considering 1 extra compliant state. It therefore becomes the fastest implementation.

In most of the remaining case studies the ENUM and EXPL implementations are the most efficient ones, both in terms of number of queries and running times. Optimisations such as enabledness dependencies or A^+/A^- sets are not free, and their benefits are realized only in the long run. In small examples these optimisations are not worth computing, and therefore the unoptimised implementations perform much better than their more sophisticated counterparts.

Manipulating declarative artefacts such as API specifications is much faster than dealing with source code. On average each satisfiability query is solved in 46 ms, while each reachability query takes roughly 1 second.

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 for instance when comparing the two versions of `MS-PCCRR`.

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 `MS-PCCRR` EPA has 10 states out of 4096.

Finally, the engine certainty was very high in all of the analysed case studies. This is remarkably high for the CVC3 and BLAST tools. Specially considering that most of the analysed subjects were relevant APIs already studied in previous work [WSCF00, DF04, AČMN05, HJM05, DKM+10].

As an overall conclusion to the quantitative data analysed in this section, we understand that EXPL^{D} dominates on the largest code examples, and is competitive everywhere else, followed closely by EXPL$^{+/-}$. This indicates that if we had to support a single algorithm (or decide on a default one for our tool) we would pick EXPL^{D}.

The enumeration-based algorithms were quite competitive on most cases, but proved to be significantly slower than their exploration counterparts in specific cases such as `Socket` or the `ATM`.

5.3. Validation Support Features

In this section we present a number of CONTRACTOR features designed to assist developers or reviewers in the task of validating software artefacts using EPAs.

5.3.1. Automatic Detection of Suspicious EPA Elements

The first set of features is related to the list of validation guidelines that we discuss later on in Section 6.4. Some of these guidelines refer to structural properties of the EPAs that can be automatically analysed. For instance, detecting the presence of a deadlock state in an EPA is straightforward: We simply check for the existence of a state whose action set is empty.

In particular, the current version of CONTRACTOR implements the following structural detection features:

- Detection of missing actions. We check the EPA to see if there exists an action $a \in Act$ that is not enabled in any abstract state.
- Detection of enabled actions with missing transitions. This is the case when the EPA features a reachable abstract state A and an action $a \in A$ has no outgoing transitions from that state.

  This occurs, for instance, in pre/postcondition specifications in which the postcondition for A is semantically equivalent to false. In such cases, no outgoing transition can ever be found since the action is not executable.

In any of these cases, CONTRACTOR will issue a warning in order to inform the user.

5.3.2. EPA Exploration Features

As we discuss later in Chapter 6 some EPAs can become too large to handle without proper tool support. In those cases, our experience indicates that guided exploration can help us tame the scale of the resulting abstraction.

In particular, in the context of A. Tcach's M.Sc. thesis [Tca10], we designed and implemented an EPA exploration extension for CONTRACTOR.

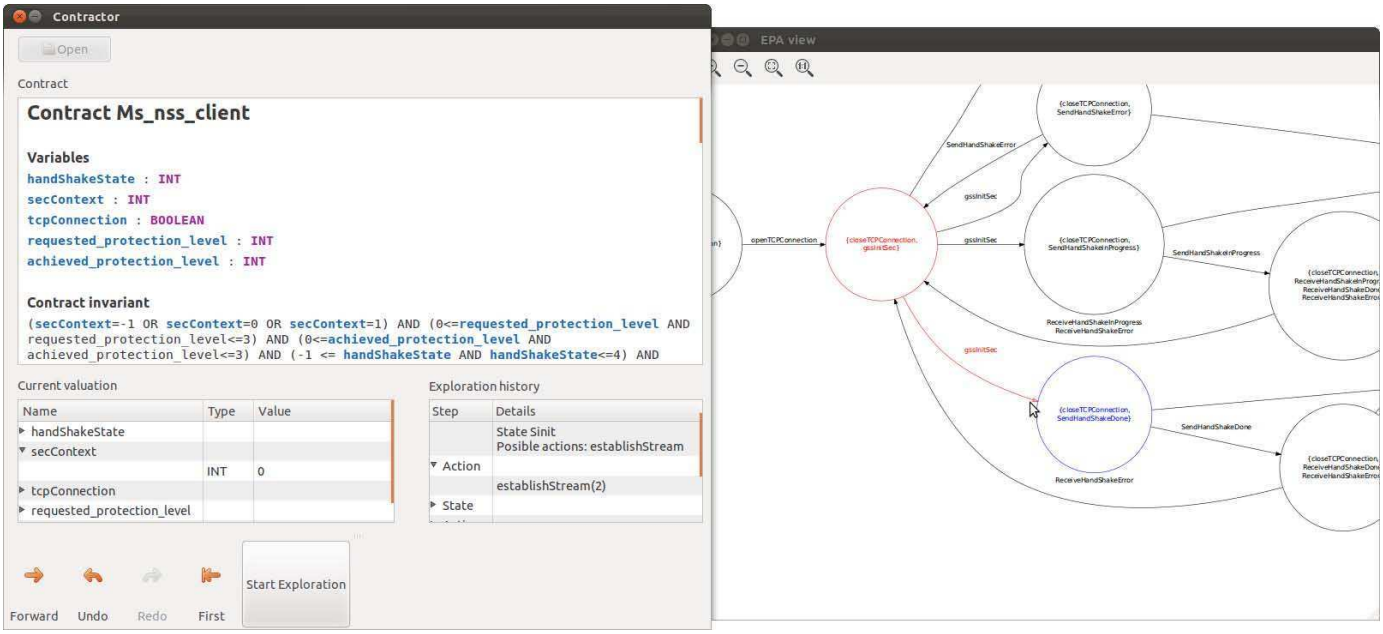

Figure 5.3: The CONTRACTOR explorer in action on the `MS-NSS` protocol

Figure 5.3 presents a screenshot of this extension. As we can see, the user is presented with two windows. On the left, the main window displays the action system details. The right window presents a graphical representation of the EPA.

The tool provides two exploration modes. In the *concrete exploration* mode the user starts from one of the initial abstract states. For each step, the user picks one of the enabled actions in the current state, together with concrete parameters for it. The graphical representation on the right highlights the current state, while the main window shows the exploration history. At any moment the user can undo or redo her steps.

In the *symbolic exploration* mode the user constructs a parameterless trace on the EPA. Notice that such trace not only prescribes the actions to be performed on the action system, but also the desired abstract destination states. The tool checks if the given trace is feasible. In other words, it tries to find concrete parameters for each action in the trace.

The reader is referred to [Tca10] for the implementation details of these exploration modes. Currently, due to limited support from the underlying decision engines, only API specifications can be explored.

5.3.3. Refining the EPA States

In our experience, grouping concrete instances according to the actions that they enable provides a good compromise between abstraction size and precision. However, in many cases the resulting abstraction is too coarse and rather uninformative.

For instance, let's consider the singly-linked list implementation first presented in Figure 2.5. The EPA helped in identifying a problem in the `remove` operation, but it is not precise enough to let a reviewer identify if elements are added at the beginning or the end of the list.

CONTRACTOR provides a feature to extend action systems with extra predicates [dCBGU12b]. These predicates are used, together with the requires clauses, when partitioning the abstract state space.

For instance, we can add an extra predicate to the action system:

$$\texttt{evenFirst} \stackrel{\text{def}}{=} \texttt{l != null \&\& l->size > 0 \&\& l->first->data \% 2 == 0}.$$

This predicate states that the list is not empty and that its first element is even.

When executing contractor on the extended action system extended with the `evenFirst` predicate, we obtain the EPA depicted in Figure 5.4.

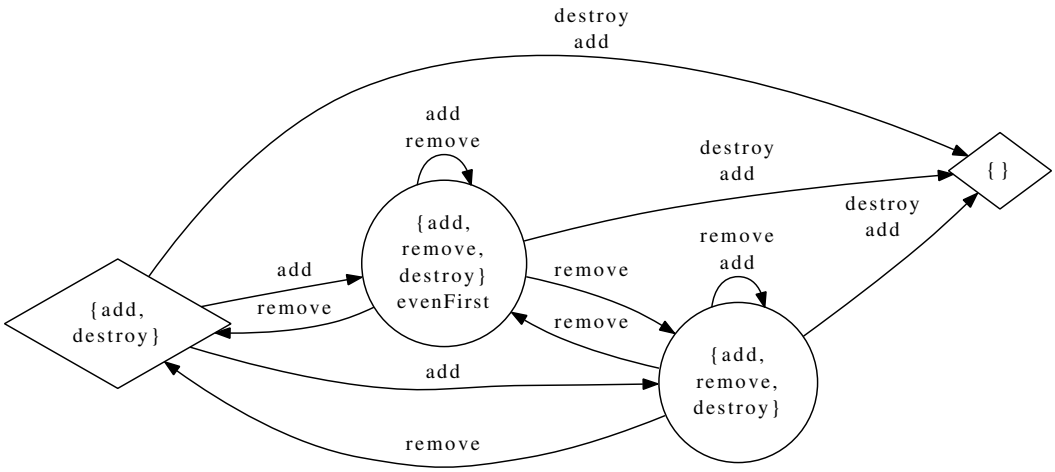

Figure 5.4: EPA for the singly-linked list with extra predicate

Notice how this EPA is richer than the original EPA presented in Figure 2.6. In particular, the abstract state `{add,remove,destroy} evenFirst` groups non-empty lists

whose first element is even. On the other hand {add,remove,destroy} groups non-empty lists whose first element is odd. These two states are not interconnected via `add` labels. This means that whenever we add an element, the parity of the list head is not affected. As a consequence, we infer that elements must probably be added on the tail of the list. On the other hand, these states are interconnected via `remove` labels, so removing elements is certainly performed on the head of the list.

In [dCBGU12b] we present a walk-through example of state refinements via extra predicates.

From an implementation perspective, predicates are treated as non-executable actions. This allows us to reuse all the machinery that we had for actions (enabledness dependencies, state predicates, etc.) without disrupting the rest of the tool internals.

5.3.4. Refining the EPA Transitions

In the previous section we discussed how sometimes adjusting the EPA abstract states may help the reviewer discover new interesting facts about the API under analysis. An analogous situation occurs when an action appears in several transitions and the reviewer may wish to further refine these to get a better understanding.

Let's revisit the singly-linked list example with the extra predicates. We discussed how the two states characterising non-empty lists where connected by means of `remove` transitions. However, they also present looping `remove` transitions of their own.

Using extra *status predicates* we can refine the `remove` operation to distinguish two cases: an odd element is being removed (retVal % 2 != 0) or an even element is being removed (retVal % 2 == 0)

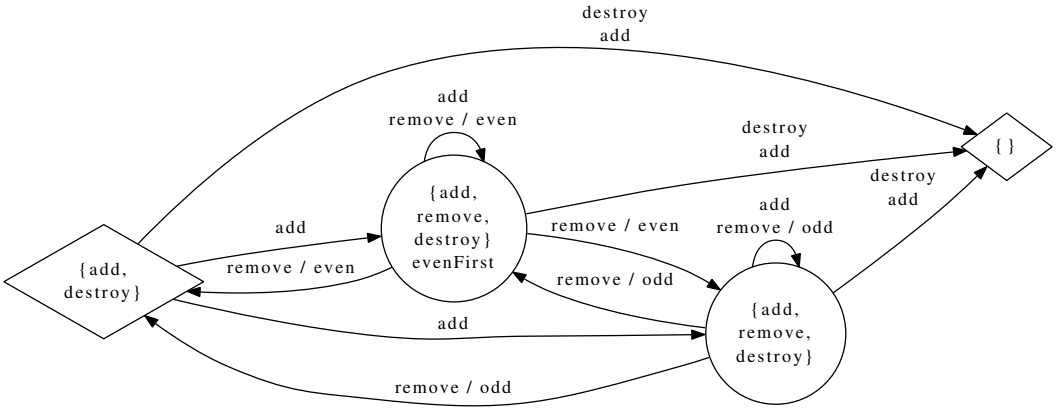

Figure 5.5: EPA for the singly-linked list with extra predicates and refined `remove` action

The refined EPA is presented in Figure 5.5. This feature, reminiscent of Mealy machines [HMU07], and its application to support validation are further discussed in [dCBGU12b].

Part III

Empirical Evaluation and Experiences

CHAPTER 6

Validation using EPAs

In this chapter 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. 2:** Is the enabledness level of abstraction useful for validating software artefacts and identifying findings that relate to bugs in code and problems in expected or documented requirements?

Notice that this research question deliberately omits the problems that arise when actually constructing EPAs: scalability, uncertainty in providing answers to the algorithms queries as these aspects were already discussed in the previous chapter when we answered R.Q. 1.

That said, in this chapter we focus in providing evidence that supports our claim that EPAs are a valuable aid during manual inspection/validation tasks. We first present a series of case studies in which EPAs were used to uncover interesting findings such as bugs, inconsistencies or undocumented behaviour. We also motivate the selection criteria for these subjects. We then dive into the findings discovered when using the EPAs as validation companions. We close this chapter presenting a series of guidelines that can assist developers/reviewers that want to use EPAs as validation aids.

The validation of our approach is continued in the next chapter, where we pose complementary research questions that evaluate how expressive and understandable EPAs are.

6.1. Experimental Setting

In order to answer the previous research question we conducted a series of case studies using each of the APIs presented in Section 5.2.1. These case studies were conducted using the following design. First, we obtain an enabledness-preserving abstraction for each API under analysis. 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 state predicates

create between EPA transitions and states with fragments of the original artefact such as requires clauses, each finding is manually tracked back to the original artefact in order to confirm it is non spurious.

The behaviour requirements, which were compared to the enabledness-preserving abstractions, were obtained as follows. The `WebFetcher` EPA was compared against its original typestate introduced in [DF04]. The `ATM` EPA was compared against its statechart presented in [WSCF00]. Both the `MS-NSS` and `MS-WINSRA` protocols were compared against the informal diagrams presented in their respective official technical documentation [MS-08, MS-09b]. 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. [DKM+10]. 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 [HJM05]. The `MS-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. [DKM+10], as well as the SMTP Protocol RFC[1].

6.2. Findings in API Specifications

6.2.1. WebFetcher

The purpose of this case study was to compare the enabledness-preserving abstractions automatically constructed by our approach with manually constructed abstractions aimed at static-time reasoning about programs. We considered a case study presented in [DF04] which extends the notion of typestates for object oriented languages: a class modelling a web page fetcher. The class provides methods to set the target URL, to open and close the connection and to fetch data, as described in Figure 6.1.

WebFetcher
variable *site* string
variable *cxn* socket
inv $site \neq \text{null} \ \wedge \ (cxn \neq \text{null} \ \Rightarrow \ cxn.state = \text{open})$
start $site \neq \text{null} \ \wedge \ cxn = \text{null}$
action `setSite`(string *s*)
pre $s \neq \text{null} \ \wedge \ cxn = \text{null}$ **post** $site' = s$
action `open()`
pre $cxn = \text{null}$ **post** $cxn' \neq \text{null} \ \wedge \ cxn'.state = \text{open}$
action `close()`
pre $cxn \neq \text{null}$ **post** $cxn' = \text{null}$
action `getPage()`
pre $cxn \neq \text{null}$ **post** true

Figure 6.1: Specification of a web page fetcher

From the specification in Figure 6.1 we derived an action system and automatically constructed the EPA depicted in Figure 6.2. The states, the transitions (as depicted in the diagram) and the state predicates (as computed according to Def-

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

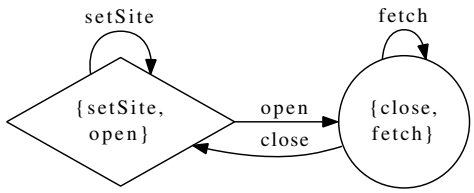

Figure 6.2: EPA for the web page fetcher

inition 3.5) that our technique produces coincide with the manually constructed typestate FSM diagram shown in [DF04].

The results of this case study support the conjecture that enabledness-preservation provides an abstraction level that is close to the level at which developers find convenient to describe protocols and API expected usage. In addition, the case study provides some indication that the automated EPA construction technique here presented could be used to produce typestates, in the sense of [DF04], automatically from specifications.

6.2.2. ATM

In this case study, our aim was to apply our approach to validate an existing contract specification produced by a third party. We took the ATM case study described in [WSCF00] where a statechart [Har87] model is inferred from scenarios and pre/post conditions for actions appearing in them. The resulting statechart can simulate the scenarios and has an invariant for each of its states based on the pre/post conditions of actions.

ATM
variable $cardIn, cardHalfway, passwdGiven$ boolean
variable $card$ card
variable $passwd$ int
inv true
start $\neg cardIn \ \wedge \ \neg cardHalfway \ \wedge \ \neg passwdGiven \ \wedge \ card = \text{null} \ \wedge \ passwd = 0$
action `insertCard`(card c)
pre $c \neq \text{null} \ \wedge \ \neg cardIn$ **post** $cardIn' \ \wedge \ card' = c$
action `enterPassword`(int p)
pre $p \neq 0 \ \wedge \ \neg passwdGiven$ **post** $passwdGiven' \ \wedge \ passwd' = p$
action `takeCard()`
pre $cardHalfway$ **post** $\neg cardHalfway' \ \wedge \ \neg cardIn'$
action `displayMainScreen()`
pre $\neg cardHalfway \ \wedge \ \neg cardIn$ **post** true
action `requestPassword()`
pre $\neg passwdGiven$ **post** true
action `ejectCard()`
pre $cardIn$ **post** $\neg cardIn' \ \wedge \ cardHalfway' \ \wedge \ card' = \text{null} \ \wedge \ passwd' = 0$
action `requestTakeCard()`
pre $cardHalfway$ **post** true
action `canceledMessage()`
pre $cardIn$ **post** true

Figure 6.3: Specification of an ATM (extracted from [WSCF00])

Using the pre/post specification in Figure 6.3 we derived an action system and obtained the EPA in Figure 6.4. Notice that [WSCF00] also provides a series of

scenarios for the ATM. We did not use these scenarios while constructing the EPA.

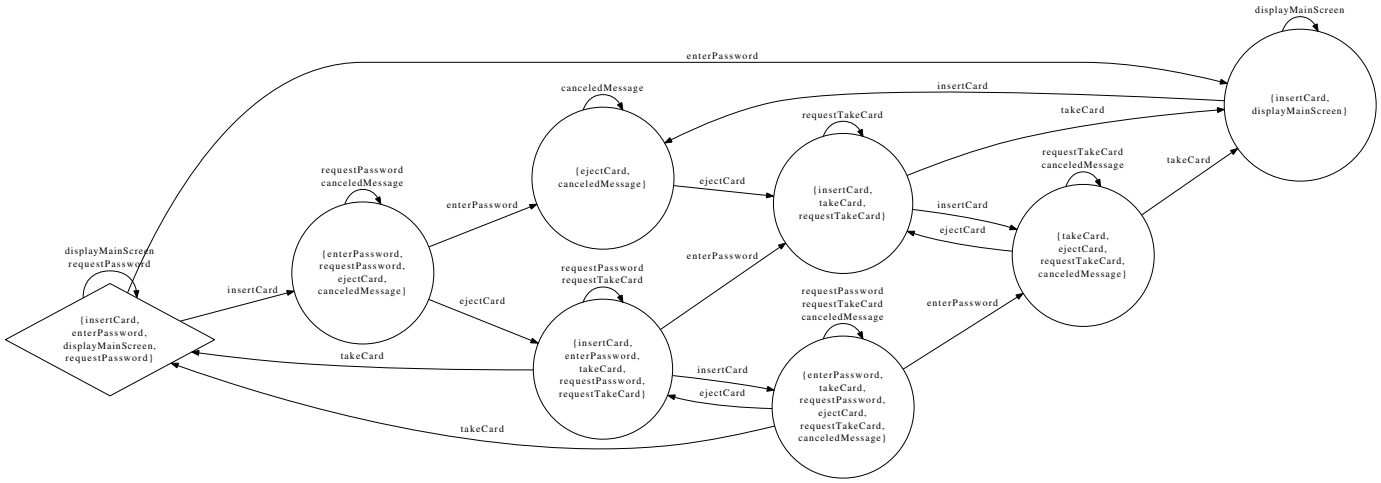

Figure 6.4: EPA for the ATM

We then compared the EPA with the statechart provided in Figure 11 of [WSCF00] with respect to simulation [Mil80]. As a result, we found that the following trace is valid in the statechart, but not valid in the EPA: `displayMainScreen`, `insertCard`, `requestPassword`, `enterPassword`, `canceledMessage`, `ejectCard`, `requestTakeCard`, `takeCard`, `displayMainScreen`, `insertCard`, `requestPassword`.

Analysis of the execution of the trace on both models, showed that while the `takeCard` action in the statechart led back to its initial state, this did not occur in the EPA. Based on this observation, we compared the state predicates reached by the execution up to `takeCard` in the EPA and the statechart. We found that they differed on the acceptable values for the *passwdGiven* system variable. In the predicate for the EPA state, *passwdGiven* is required to be true, while in the predicate for the statechart state, *passwdGiven* is required to be false. Further analysis shows that `takeCard`'s postcondition in Figure 6.3 does not update the *passwdGiven* system variable to false. The impact of this omission is that, according to the pre/post specification in Figure 6.3, the ATM never returns to a state where it can accept a new password to be entered because it already has one. In addition, it shows that the synthesis algorithm in [WSCF00] does not guarantee preservation of postconditions in the synthesised statechart.

In summary, the construction of an EPA from the pre/post specification of an ATM in [WSCF00] supported uncovering errors in the specification and problems with the actual synthesis algorithm therein proposed.

6.2.3. .NET NegotiateStream Protocol

The aim of this case study was twofold. On one hand, we intended to validate the utility of the approach in aiding the construction of pre/post condition-based specifications. The hypothesis was that by using behaviour models early in the development of the specification, bugs can be detected and guidance on how to fix them can be obtained. On the other hand, we aimed at validating whether the approach can support identifying problems in real specifications.

Using the quality process and model-based testing approach described in [GKM+08b] as a starting point, we selected a Microsoft protocol specification currently under revision: The MS-NSS protocol [MS-08] conceived for the negotiation of credentials between a client and a server over a TCP stream.

The protocol has two phases: *i)* a negotiation phase in which client and server exchange security tokens using the GSS-API [Lin93] and *ii)* a data transfer phase in which actual data is transmitted according to the negotiated standards.

Basically, the negotiation phase starts with the client sending a security token to the server including a requested security level (e.g., encryption and/or signature). The server processes this token and sends an answer to the client, which processes it and sends back another answer. This process is repeated while the token that they send each other is a *continuation token* and is finished usually when one of the following situations takes place:

- An error message is sent by either the client or the server, in which case the client may try again or terminate the negotiation.
- The server sends an acceptance token indicating to the client the end of the first phase (a security mechanism like Kerberos may have been sucessfully negotiated). This token includes the final protection level, which could be weaker than the required by the client.

Once the data transfer phase begins, the client can exchange data with the server. Data exchange requires framing when signature and/or encryption are implied by the negotiated protection level. As in the negotiation phase, the data exchange phase can result in an error in which case the communication is usually terminated.

The experimental setup for this case study (which can be seen diagrammatically in Figure 6.5) was as follows. First, a person completely unfamiliar with the protocol but experienced in writing pre/post condition-based specifications read the publicly available technical documentation describing the protocol [MS-08]. Then, the same person wrote a formal specification for the protocol validating the protocol against the document. Once the protocol's specification was completed, an engineer from our team with experience in protocol validation analysed its EPA in order to validate the specification and the protocol tecnical documentation.

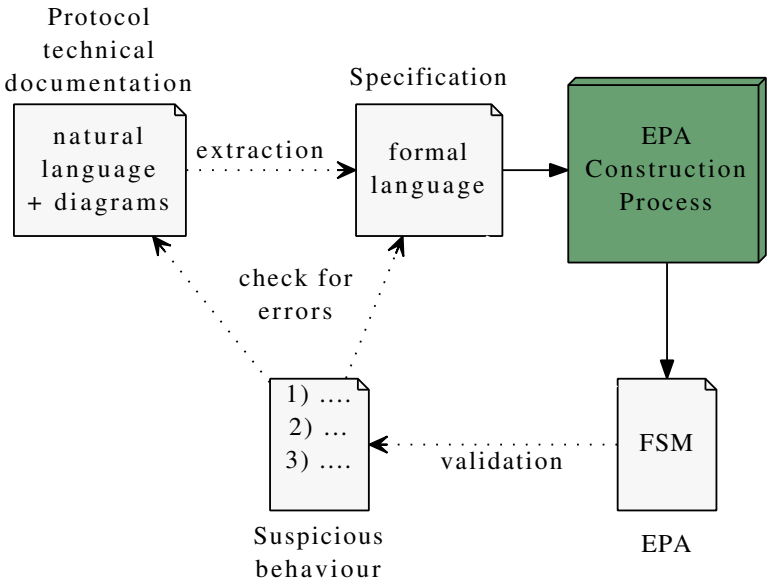

Figure 6.5: Experimental setup for the MS-NSS case study

The protocol's technical documentation is structured natural language description containing two auxiliary state machines. The documentation states that the

natural language description is to be considered the normative specification of the protocol while the state machines are simply references for the reader.

The inferred protocol specification included a set of controllable and observable actions appearing in the technical documentation of the client side of the protocol. Only the information provided in natural language was used as a source for the specification. For instance, Figure 6.6 depicts a natural language fragment of the original technical document, together with its specification translation. It is worth mentioning that models developed in [MS-08] include server and client-side requirements since the main goal of the QA project is to check protocol technical documentation compliance against Windows products. For our experiment, the modeller only referred to the client-side specification section of the document.

"*If the* `gss_init_sec_context` *function returns an error code, then the client MUST create a* `HandshakeError` *message, placing the returned error code in the* `AuthPayload` *of the messages as described in section 2.2.1.*"

action `sndError()`
pre tcpConnection ∧ handShakeState = Proccessed ∧ gssReturned = Error
post (handShakeState' = NotStarted ∧ tcpConnection') ∨ (handShakeState' = Error ∧ ¬tcpConnection')

Figure 6.6: MS-NSS documentation fragment and corresponding translation

During the specification development process the modeller iteratively constructed EPAs in order to use them as an aid to eliminate bugs and typos from the specification being developed. The EPA was analysed using: *i)* inspection techniques, *ii)* simulating scenarios appearing in the protocol specification document, *iii)* checking for bisimilarity of the EPA against the auxiliary client side state machine of the protocol specification document, and *iv)* composing the EPA in parallel with the the server side auxiliary state machine of the protocol specification document. Such analyses uncovered inconsistencies in the specification-under-development such as a client trying to send a token before having produced it, or a client receiving responses to messages that had never been sent to the server. As a result of the construction effort, a number of under-specified aspects were identified in the protocol's technical documentation (these were documented and modelled as non- deterministic actions in the specification).

The validation of the final specification and, indirectly, of the protocol technical documentation was performed by the experienced engineer. Most of the validation was done by inspection, guided by the enabledness-preserving abstraction (Fig.6.7) and the modeller's expertise, going into the detail of the specification and finally the protocol technical documentation if needed.

As a result of this final validation by the experienced engineer, three kinds of issues arose. First, two questions regarding the behaviour of the client were raised. These issues point to potential problems in the protocol technical documentation:

- In abstract state {`closeTCP`,`rcvDone`,`rcvError`} of the EPA, the client has just sent a message to the server indicating that the negotiation phase is over (`sndDone`). However, at this point the server could potentially reply with a continuation token (`rcvInProgess`), which the client cannot accept in that abstract state. From the document it is not clear what should happen to this continuation token.

- Abstract states {`sndError`,`closeTCP`}, {`closeTCP`,`rcvDone`,`rcvError`} and {`closeTCP`,`rcvInProgress`,`rcvDone`,`rcvError`} have outgoing transitions with error labels that go to both the initial and the

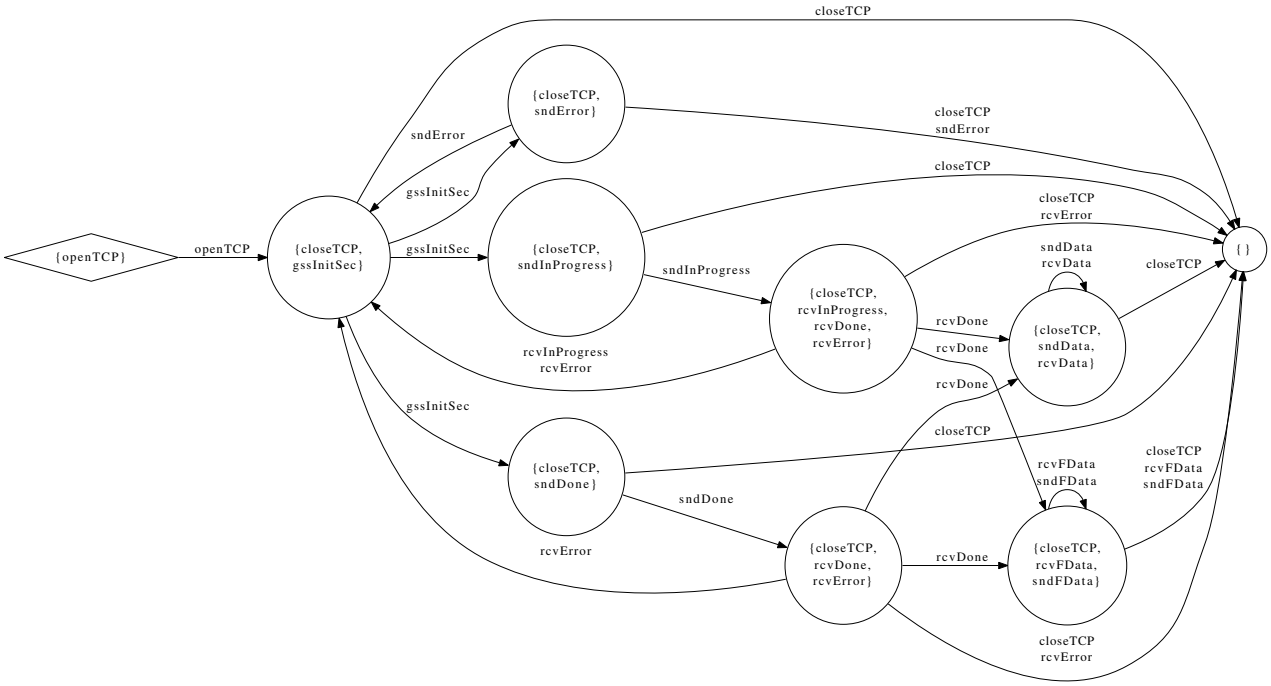

Figure 6.7: EPA for the NegotiateStream protocol

deadlock state. This non-determinism reflects underspecified behaviour described in the protocol specification document. However, this underspecification seems to be problematic as an implementation could decide unilaterally whether to (return to the initial state and) try to reuse the connection despite the error or to deadlock and require the user to restart with a new protocol instance. However, the server side does not seem to be prepared for such a non-deterministic choice on the client side.

Second, an inconsistency between the natural language specification of the client behaviour and the state machine of the protocol specification document describing the server behaviour was identified: The EPA for the client constructed automatically from the protocol's specification composed in parallel with the state machine for the server leads to a deadlock. A trace to the deadlock, raises the following question:

- The EPA shows that the specification allows a client to receive `rcvDone` without ever sending a `sndDone` message. This implies that the server may unilaterally decide to enter the data transfer phase, which leads to a deadlock. Why is the client not sending a `sndDone` message before being allowed to receive `rcvDone`?

Note that, in fact, the inferred formal specification and hence the natural language specification for the client is consistent with the natural language description for the server (which is the normative part of the technical documentation), hence the issue raised above actually shows a discrepancy between text specification and diagrammatic-aid of the server side in the protocol technical documentation.

Finally, inspection of the EPA and comparison against the auxiliary client side state machine of the protocol technical documentation helped find some discrepancies between the textual description and the diagrammatic aid for the client side:

- In the state machine of the protocol technical documentation, action `sndError` goes to a state in which the client waits for a message from the server. However, the EPA shows that after this event the client should either terminate the connection or retry the whole phase. The EPA is consistent with the protocol technical documentation text.

- Analogously to `sndError`, when the state machine of the protocol technical documentation for the client side receives `rcvError`, the client must wait. However, the EPA, in agreement with the protocol technical documentation text, shows that this is not the case.

Some of the issues reported above, for version 2.0 of the protocol technical documentation that was available at the time, were subsequently corrected in version 3.0 of the document. This shows that the issues identified were not only real but also relevant enough to warrant correction.

In summary, in this case study, the automated construction of an EPA for an industrial strength document aided significantly in: *i)* correcting and elaborating a formal specification of the protocol, in *ii)* identifying relevant problems in the pre-existing real natural language (and auxiliary diagrammatic) protocol technical documentation, and *iii)* in automatically constructing a diagrammatic aide which is sound with respect to the protocol technical documentation (as opposed to manually generated diagrams that have inconsistencies with the normative description of the protocol).

On a final note, it is worth mentioning that the EPA in Figure 6.7 featured almost the same level of abstraction as the state machines in the protocol technical documentation. The main differences were that the EPA has more states and transitions because it models local GSS-API calls explicitly and distinguishes encrypted and plain data transmissions. This similarity in abstraction level is not only relevant because it allows validations based on bisimulation and parallel composition of artifact but also because it supports the conjecture that enabledness-preservation provides an intuitive abstraction level that is close to the level at which developers describe protocols and API expected usage.

6.2.4. WINS Replication and Autodiscovery Protocol

The purpose of this case study was to analyse the limitations of our approach that arise from dealing with a large industrial strength contract specification. These difficulties are mainly divided in two categories: scalability of the construction algorithm in terms of time and memory consumption and feasibility of validating the output EPA which has the potential of having billions of states. Scalability concerns were discussed in Section 5.2; we discuss the validation feasibility next.

We chose another Microsoft protocol specification, in this case the one for the "WINS Replication and Autodiscovery Protocol" [MS-09b]. This protocol, also known as WINSRA, governs the process by which a set of name servers discover each other and share their records in order to keep an up-to-date vision of the name mappings.

A name server can have two different roles when interacting with other servers. It can be in *pull replication* mode, in which, from time to time, the server asks its partners whether they have something new, and then fetches the differences between its own name mapping and that of its partners. Or it can be in *push replication* mode, in which it informs its partners that there is some new information that they need to be aware of, so they can fetch it.

On a pull replication round, a name server goes through the following actions:

1. It initiates network traffic, indicating the replication mode with `initiateTrafficPull`.
2. It establishes an association with its partner using `associationStartRequestControlSuccess`. Once the request is sent it awaits for an `associationStartResponseObserve` response.
3. Once the association is set up it requests a mapping indicating which are the maximum and minimum version numbers for each server having name records owned by its partner with `ownerVersionMapRequestControlSuccess`. It then waits for its partner to send this mapping via `ownerVersionMapResponseObserve`.
4. Once it has the versions mapping it calculates which name records it needs to update and proceeds to request them one by one with successive `nameRecordsRequestControlSucce` Each of these messages has its corresponding `nameRecordsResponseObserve`.
5. Finally, when there are no more name records that need to be requested, it finishes its association by sending `associationStopRequestControlSuccess`.

The push replication round is symmetrical:

1. The round starts with traffic initiation, which is performed with `initiateTrafficPush`.
2. Once the traffic has been initiated, the name server waits for its partner to connect and send an association start request with `associationStartRequestObserve`. Once received, this request is answered with an `associationStartResponseControlSuccess`.
3. An `updateNotificationControl` action happens in which the partner is sent the mapping (as if it had been requested).
4. The name server expects its partner to ask for name records with `nameRecordsRequestObserve`. Each of these record requests is responded with `nameRecordsResponseControlSuccess`.
5. Finally, a disconnection from the partner is expected with `associationStopRequestObserve`.

Notice that this brief description of the WINSRA protocol is simplified for the sake of presentation. The actual protocol deals with the fact that each participant can switch between the push and pull roles in particular situations, as well as being able to act as both pull and push partner at the same time. There also exists the possibility for the pull partner to act in "data verification" mode, which adds complexity.

The case study was conducted as follows (refer to Figure 6.8 for a diagrammatic representation of the process): The initial documentation available was, as with the previous case study, a publicly available protocol specification document [MS-09b] including a normative natural language description of the protocol together with diagrammatic aids in the form of state machines, and a SpecExplorer [CGN+05] model of the protocol. The SpecExplorer model had been created by a different team than the one that developed the protocol specification.

We used the SpecExplorer model as the basis for constructing a protocol specification in the form of an action system that could be input to our approach. A systematic translation procedure was used for translating the SpecExplorer model into a specification: a variable was created for each of the SpecExplorer model variables and one action for each method in the SpecExplorer model. For action's requires clauses we used the REQUIRES clauses of the SpecExplorer model and for functions of each action we performed manual strongest postcondition calculus.

Notice that the SpecExplorer model is 2500 lines long, featuring a class with 16 fields some of which are complex data types such as maps and sets. This class

describes 33 actions, which are composed of the aforementioned events, together with special variants used in special cases. For instance, `NameRecordsResponseControlDisconnect` is used when the obtained name record was the last one and the partner proceeds to disconnect.

Surprinsingly, the action system invariant for this specification was almost trivial as it only required to indicate that an integer value used as a count was never negative.

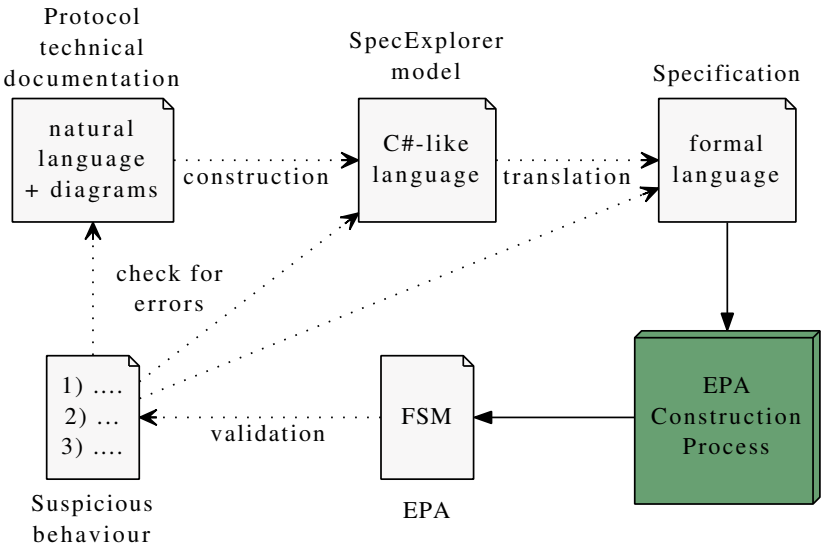

Figure 6.8: Experimental setup for the WINSRA case study

We obtained an EPA protocol specification obtained by translating the SpecExplorer model. This initial abstraction featured 60 states and 642 transitions. Various standard finite state machine analysis techniques such as hiding and minimisation were needed to handle an abstraction that had, in its initial version, 20 times more transitions than the MS-NSS protocol.

Similarly to the MS-NSS protocol case study, an iterative process was enacted in which first the protocol specification was used to produce an EPA for validation. Second, the resulting abstraction was analysed and a list of issues that were thought to be suspicious was generated. This list was then validated against the SpecExplorer model and the protocol technical documentation. Errors detected in the specification or in the SpecExplorer model were subsequently corrected and an EPA generated again.

We now reproduce parts of the validation process that led to detecting flaws in:

1. The specification with respect to the SpecExplorer model.

2. The SpecExplorer model with respect to the WINSRA technical documentation.

3. The WINSRA technical documentation with respect to the intended protocol behaviour.

From the first version of the EPA, our impression was that there was something wrong with the specification: the natural language protocol description did not seem to be describing a protocol with such a rich number of modes that would lead to a 60-state abstraction. In other words, it did not seem to be a case that the protocol

action `associationStartRequestControlSuccess()`
pre ... ∧ ((association = None ∧ persistent = No) ∨ (association = Pull ∧ replicationType = Push) ∨ (association = Push ∧ replicationType ≠ Push) ∨ association ≠ Both)
post protocolState' = AssociationStartRequestControl

Figure 6.9: Buggy specification for association start request (fragment)

actions could legally be combined in 60 different ways. We decided to inspect the states closest to the initial one in order to see if we could identify clues as to why the specification produced so many states and transitions.

The first issue that we found is that the action `setupInitialization`, which is supposed to be called once, appeared in a looping transition (i.e., a transition that has the same source and target states). We discovered that the SpecExplorer model's REQUIRES clause for this method was too weak with respect to the original protocol specification document. Having fixed the model and the specification, we obtained a new EPA, but the amount of states and transitions remained almost the same.

The next step was to discover that there was a state very close to the initial one that had more than 60 incoming transitions. Such a high fan-in was a warning sign. We observed that the actions available in that state were `initiateTrafficPull`, `initiateTrafficPush` and `initiateTrafficDataVerify`[2]. We revisited the protocol technical documentation and discovered that the protocol allows partners to establish persistent associations that last along several replication rounds. This persistent behaviour is modelled by calling traffic initiation actions in advanced stages of the protocol. In the SpecExplorer model this was allowed by letting any of the traffic initiation actions occur at any time. This was too permissive, since it is not true that a new persistent round can be initiated at any time. According to the technical documentation this is not intended to happen until a replication round is over.

Correcting this issue in the model (and, by translation, in the specification) involved modifying the REQUIRES clause of the three traffic initiation methods. After this change, the EPA had 54 states and 467 transitions.

We further analysed the EPA states nearest to the initial one and found that, even having fixed the traffic initiation process, the association start request and observe actions were creating another high fan-in state (of about 30 incoming transitions). We carefully inspected the REQUIRES clauses for those methods in the SpecExporer model and found two errors:

1. The REQUIRES clauses for these methods enumerate a series of conditions as the one in Figure 6.9. This was suspicious, since a valuation with values association = Pull and replicationType = No would be accepted, even when it is clear that replicationType is not Push. The conditions were in fact wrong, and corrected by replacing the conjunctions with logical implications and the disjunctions by conjunctions.

2. The REQUIRES clauses for these methods were lacking a condition over the variable *protocolState* which is used throughout the protocol life to indicate in which stage the protocol currently is in (This is basically achieved by keeping record of which was the last received or sent message). A correction was introduced by indicating that in order to start an association (or observe an

[2] `initiateTrafficDataVerify` is used for a scheduled data verification process that is similar to the standard pull replication process.

association request) the previous action must have been a traffic initiation and the one before that a setup initialisation action.

Having corrected this issues in the model and in the specification we obtained an EPA with 38 states and 233 transitions, which is roughly half the size than before. The following is a list of errors that we found using the EPA in this iteration:

- The `setupInitialization` action may go to a state with no enabled actions. This was caused by a weak action system invariant, which did not account for the fact that whenever the system was not initialised, then a boolean variable was necessarily fixed to be false (more precisely, $\neg$isSetupInitialized $\Rightarrow$ $\neg$replicationOn). This was corrected.

- The REQUIRES clauses for the traffic initiation methods were allowing the server to persistently associate in push mode with a partner and then re-establish communication taking the pull role. This was corrected by making available of the traffic initiation methods in the case of the protocol beginning, in which we have not yet taken a role. Successive calls to traffic initiation methods are restricted in order to allow them only if we keep the role we already had. The SpecExplorer model and the resulting specification were corrected.

- The `updateNotificationObserve` action may go to a state in which the only available action is to end the association. This is not correct since when the push partner is telling that there is something new, then the following action is a name records request. This anomalous behaviour was due to an error in the specification translation from the SpecExplorer model. In particular, the `updateNotificationObserve` action may leave a variable that indicates how many name records have to be requested with value 0 and this was too permissive.

During the validation process, we identified a number of suspicious behaviours in the EPA that turned out to be perfectly acceptable behaviour. These cases correspond to when our understanding of the technical document was incomplete or incorrect and do not highlight neither errors in the technical document, nor the SpecExplorer model, nor the translated specification. However, they do show that validation using EPAs can help in understanding complex protocol specifications. Some of the issues that led us to gain a better understanding of the protocol were:

- After the association has been just established, name records can be requested even when the owner version map has not yet been requested. This appeared to be incorrect, but we checked the technical documentation and this was possible when the protocol was in data verification mode.

- Once we get the owner version map we can directly disconnect. This also appeared to be incorrect, but in fact it is not. This can happen if the owner version map that we get indicates that the partner has nothing to offer us. In that case we can not ask for name records.

In this case study the automated construction of an EPA was helpful in: *i)* correcting and elaborating a formal specification of the protocol, and in *ii)* identifying relevant problems in pre-existing industrial formal models of the protocol specification. The case study showed that large EPAs are still amenable to analysis and helpful in finding problems in software development artifacts.

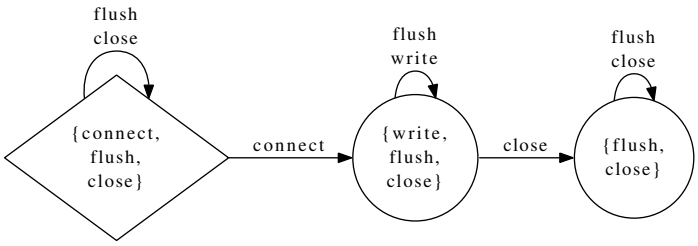

Figure 6.10: EPA of JDK 1.4 `PipedOutputStream`

6.3. Findings in API Implementations

6.3.1. 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.
- `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 constructed an action system using the JDK 1.4 implementation of the `PipedOutputStream`. 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 the system invariant.

The model in Figure 6.10 is the EPA we obtained for the `PipedOutputStream`. This abstraction is 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 `close` transition from state $\{\texttt{write}, \texttt{flush}, \texttt{close}\}$ to $\{\texttt{flush}, \texttt{close}\}$, since the latter does not allow to perform the `write` operation.

More interestingly, the abstraction of Figure 6.10 shows a `close` loop transition on the initial state, which contradicts the Java documentation since it allows the following trace: $\texttt{close} \rightsquigarrow \texttt{connect} \rightsquigarrow \texttt{write}$, 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.

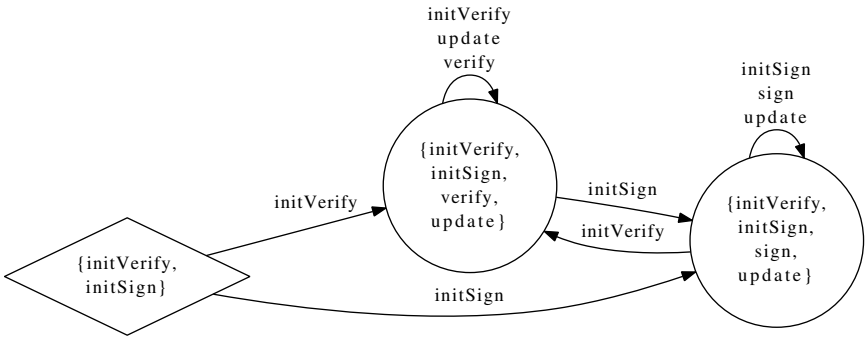

Figure 6.11: EPA of JDK 1.4 `Signature`

6.3.2. 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.
- `verify(byte[] signature)` checks the cryptographic signature given in the parameter.

As with the previous class, we constructed an action system from the JDK 1.4 implementation of the `Signature` class. We defined the requires clauses with the adequate manually extracted code fragments. We set the system invariant as true.

The model in Figure 6.11 is the EPA we obtained for `Signature`. This EPA was exactly the same as the manual model presented in [DKM+10]. 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 [DKM+10] (see Figure 6.12), as well as of the restrictions imposed by the official Java documentation.

6.3.3. 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

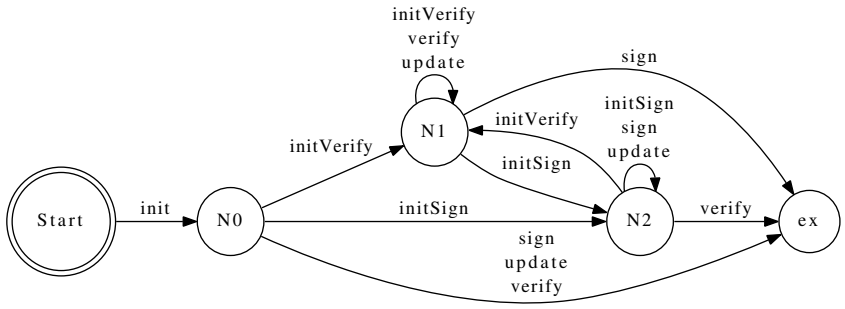

Figure 6.12: Manually generated model of JDK 1.4 `Signature` (extracted from [DKM+10]). *init* indicates the constructor, while *ex* is an error state reached when exceptions are thrown

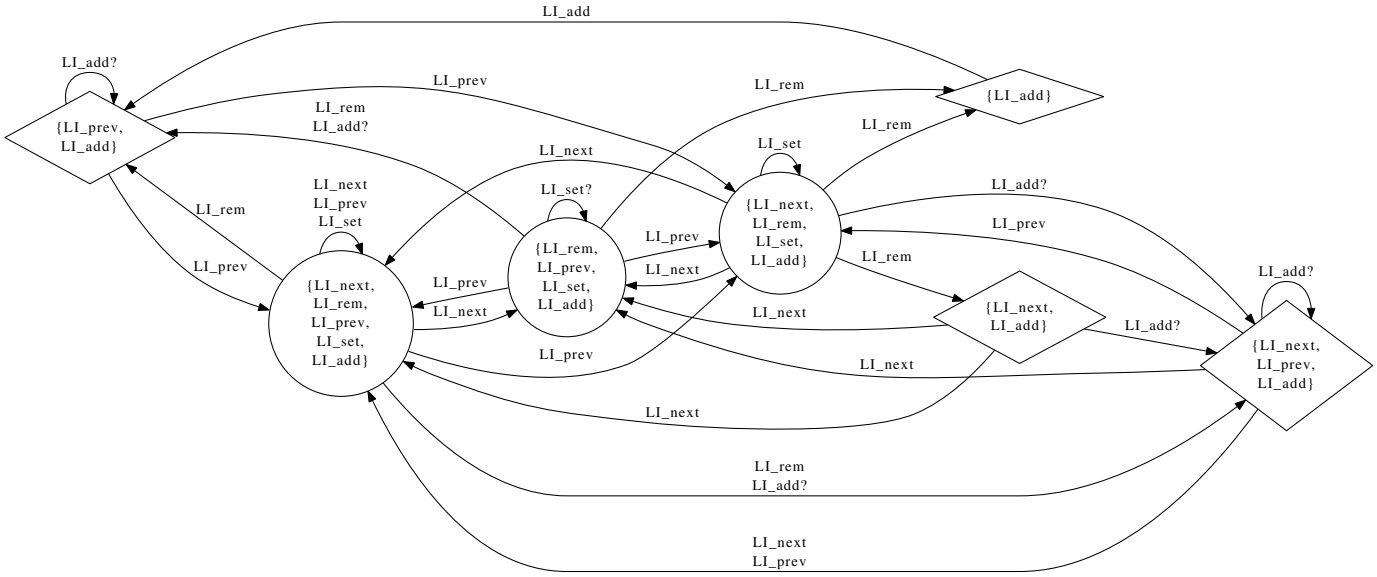

Figure 6.13: EPA of JDK 1.4 `ListItr`

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.
- `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 created an action system using the JDK 1.4 Java List Iterator implementation. The requires clauses where defined using manually extracted fragments of code so that no exception was thrown. The system invariant was manually generated and it included 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 6.13 is the EPA for the `ListItr` class. Every state in it represents an interesting situation to which an iterator can evolve. There are 4 initial states:

- {`add`}. We are iterating over an empty list since adding is the only available operation.
- {`next`, `add`}. 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.
- {`prev`, `add`}. Similar to the previous state, but since `prev` is enabled and `next` is not, the cursor is pointing at the end.
- {`next`, `prev`, `add`}. The `set` and `rem` operations are disabled, so there has not been a retrieved element. Since both `prev` and `next` are enabled, the cursor is pointing at a position in the middle of a list.

The rest of the states are similar to the states we already introduced. The only difference between them is that an element has just been returned, so the `rem` and `set` operations are also enabled.

Notice that some of the transitions in this EPA are suffixed with a "?" symbol. As we explained in Section 4.3 we use this notation when the underlying decision engine (e.g., the reachability query solver) returns an uncertain answer.

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 6.14, by analysing the JDK implementation for the list iterator. During this creation process, the developer executed a number of usage scenarios to refine his understanding of the code.

When comparing this manually-generated model with the EPA, 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.

6.3.4. 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:

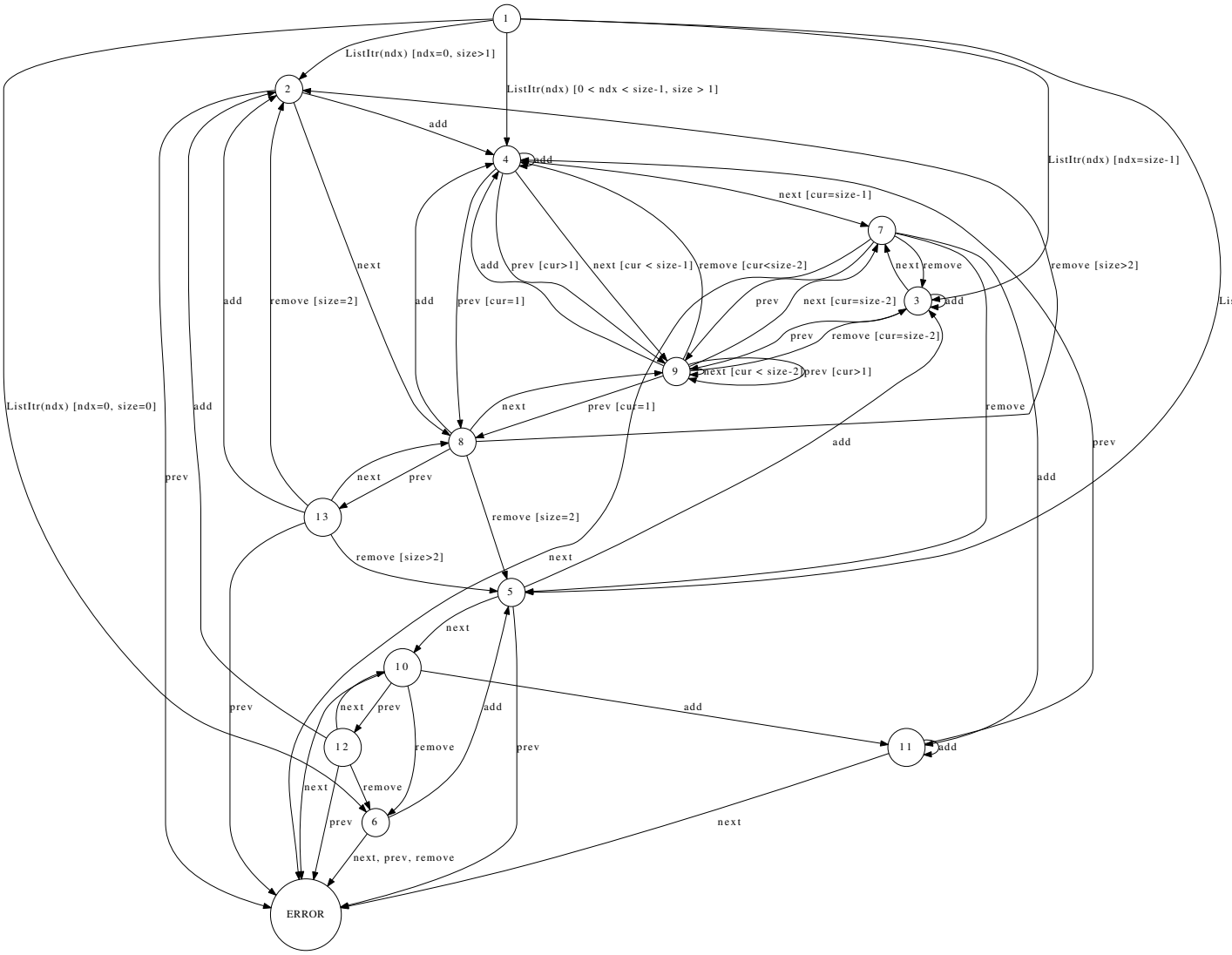

Figure 6.14: Manually generated `ListItr` behaviour model

- `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.

We constructed an action system for this class such that 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. From this action system, we obtained the EPA depicted in Figure 6.15.

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

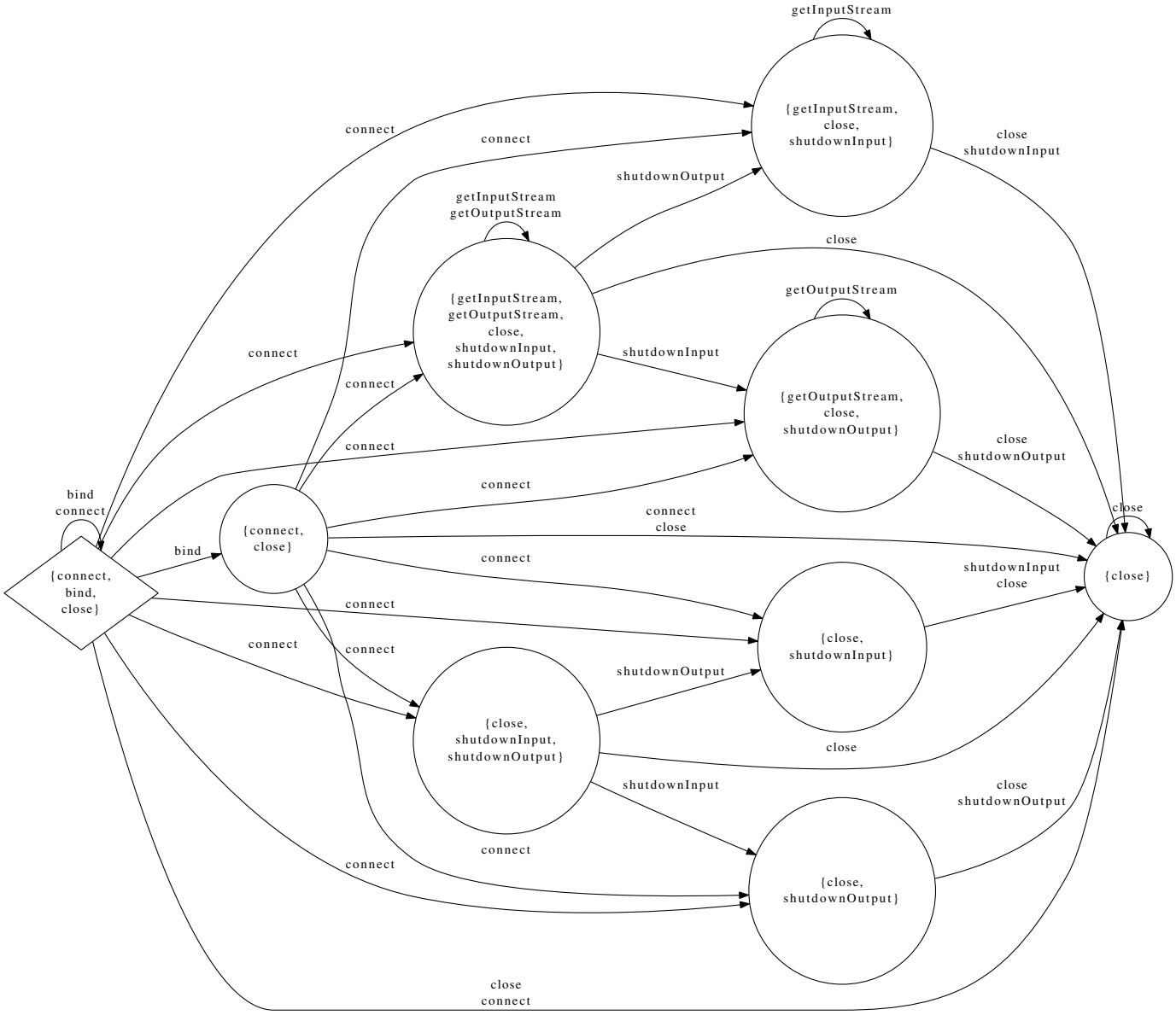

Figure 6.15: First EPA of JDK 1.4 `Socket`

{`getInputStream`,`getOutputStream`,`close`,`shutdownInput`,`shutdownOutput`}. 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 {`connect`, `bind`, `close`} and {`connect`, `close`} abstract states. The boolean variables that store whether the sending and receiving streams are closed are always `false` since the `Socket` creation. However, this restric-

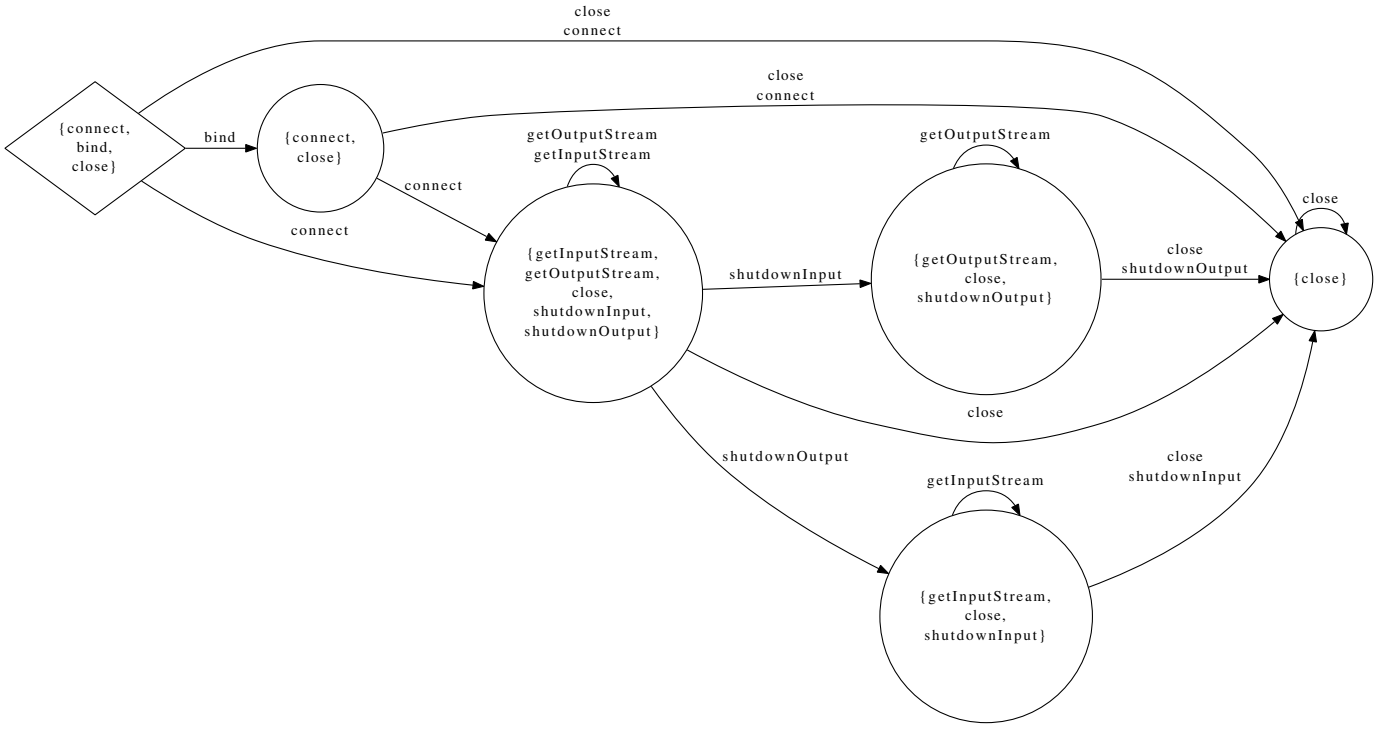

Figure 6.16: Final EPA of JDK 1.4 `Socket`

tion 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 to our tool, 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 $\{\texttt{connect}, \texttt{bind}, \texttt{close}\}$ and $\{\texttt{connect}, \texttt{close}\}$ abstract states. This new version produced the EPA in Figure 6.16.

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 [HJM05], but it requires the user to add six predicates to keep track of the state.

6.3.5. PCCR Framework

The *Peer Content Caching and Retrieval* (PCCR) [MS-09a] 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 (`MS-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 [GKSB11], 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 `MS-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:

- `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.

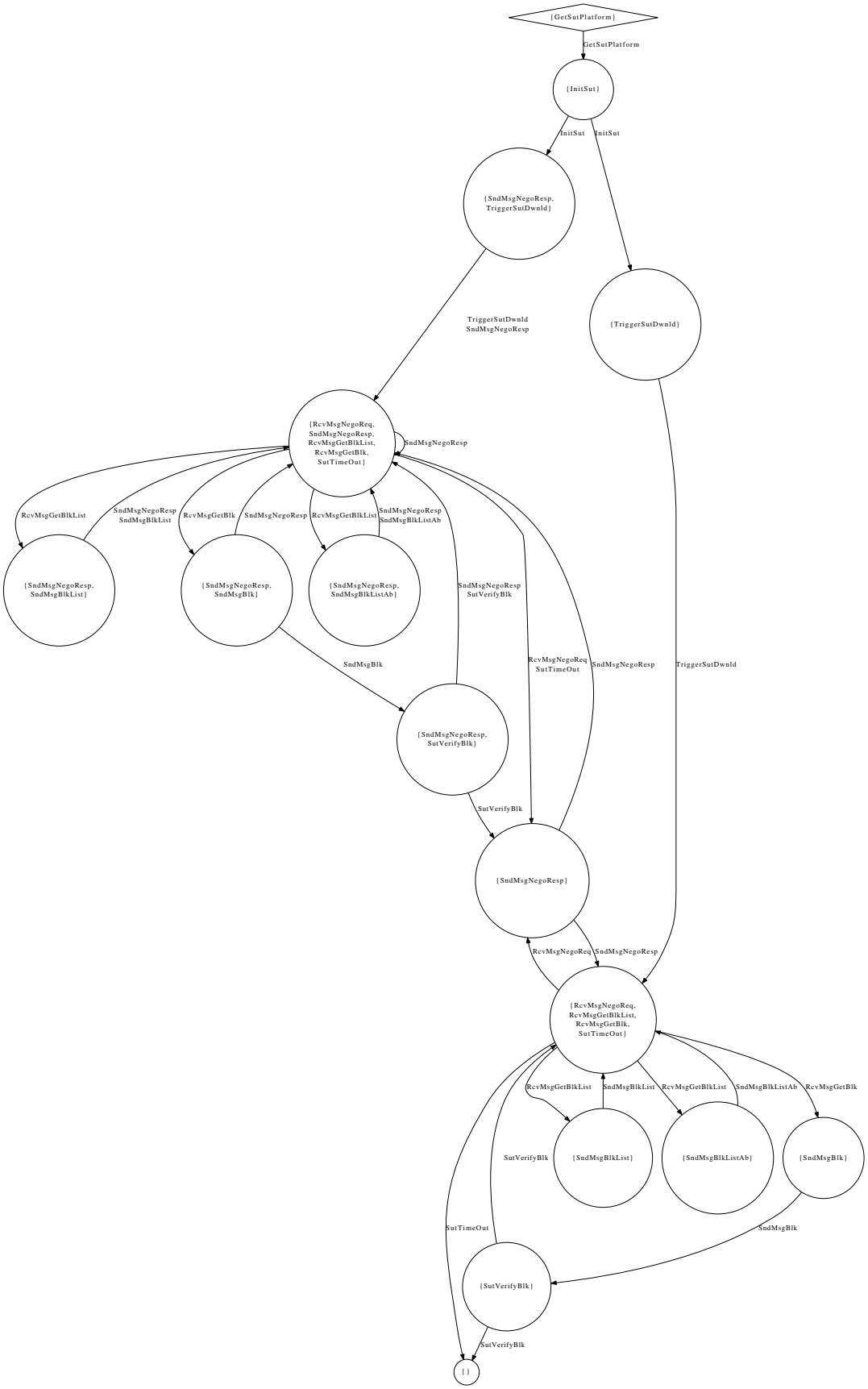

Figure 6.17: First EPA of the `MS-PCCRR` server side

We created an action system based on the C# class, using the guards for the SpecExplorer rules as requires clauses. 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 4.3.2.

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 we obtained, which had 16 states, can be seen in Figure 6.17. It was relatively big but still much smaller that the model with 844 states produced by SpecExplorer during an exploration of the `MS-PCCRR` state space. The reviewer analysed this abstraction and found that starting from the initial state the `InitSut` initialisation action showed non-deterministic behaviour, as it can evolve to states `{TriggerSutClientDownload}` and `{SendMsgNegoResp,TriggerSutClientDownload}`.

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. In the EPA depicted in Figure 6.17, this issue is manifested as the quasi-partition of the states into two sets which are only connected by 2 transitions. states `{SendMsgNegoResp}` and `{ReceiveMsgNegoReq,ReceiveMsgGetBlkList,ReceiveMsgGetBlk,SutClientReceivingTimeOut}`. 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 6.18.

We then modified the original code, 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.

6.3.6. SMTP Server

The `SMTPServer` class is a Java implementation of an SMTP protocol server extracted from `Java Email Server` (JES) [3].

- `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.

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

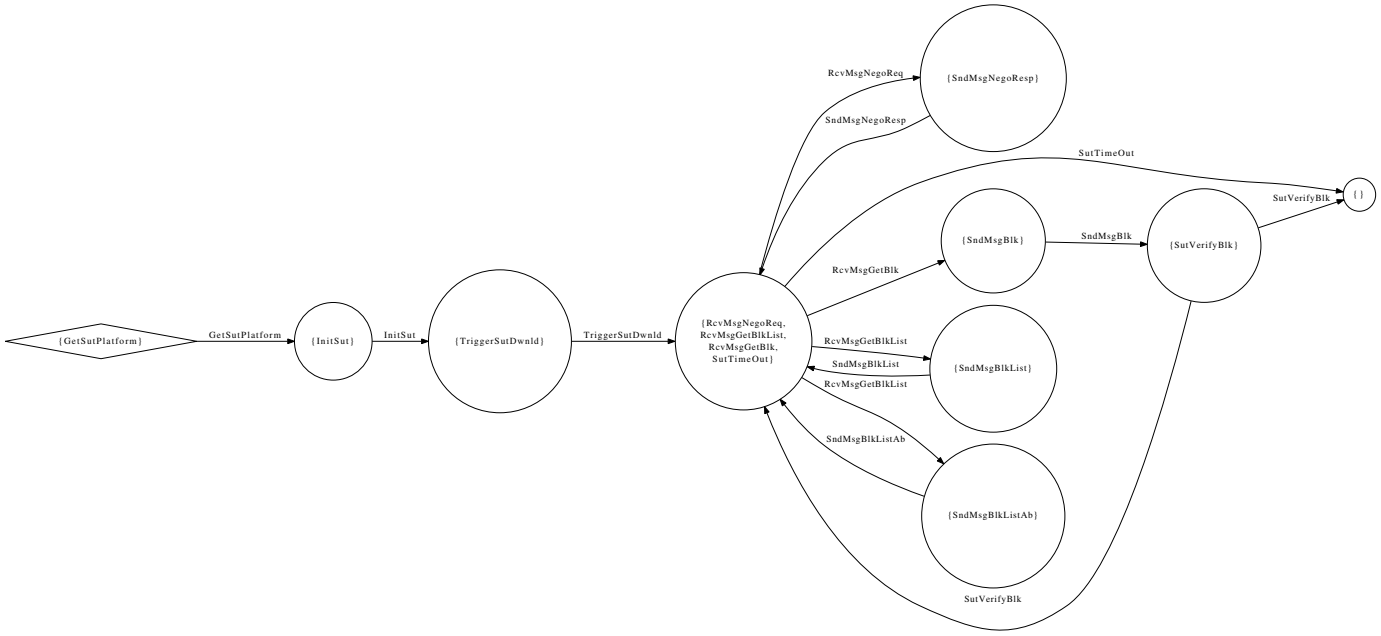

Figure 6.18: Second EPA of the `MS-PCCRR` server side (eliminated the `isTestingNego` field)

- `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.

We derived an action system from this class, 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 6.19.

We found two anomalies in the EPA:

1. After a `mail` command ({`vrfy`,`noop`,`rset`,`ehlo`,`mail`} → {`vrfy`,`noop`,`rset`,`ehlo`,`rcpt`}), a `noop` operation does not behave like it should, since it makes the EPA evolve to another abstract state. This non-empty behaviour for the `noop` operation is against the SMTP protocol standard.
2. After a `data` command ({`vrfy`,`noop`,`rset`,`ehlo`,`rcpt`,`data`} → {`vrfy`,`noop`,`rset`,`ehlo`}), 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

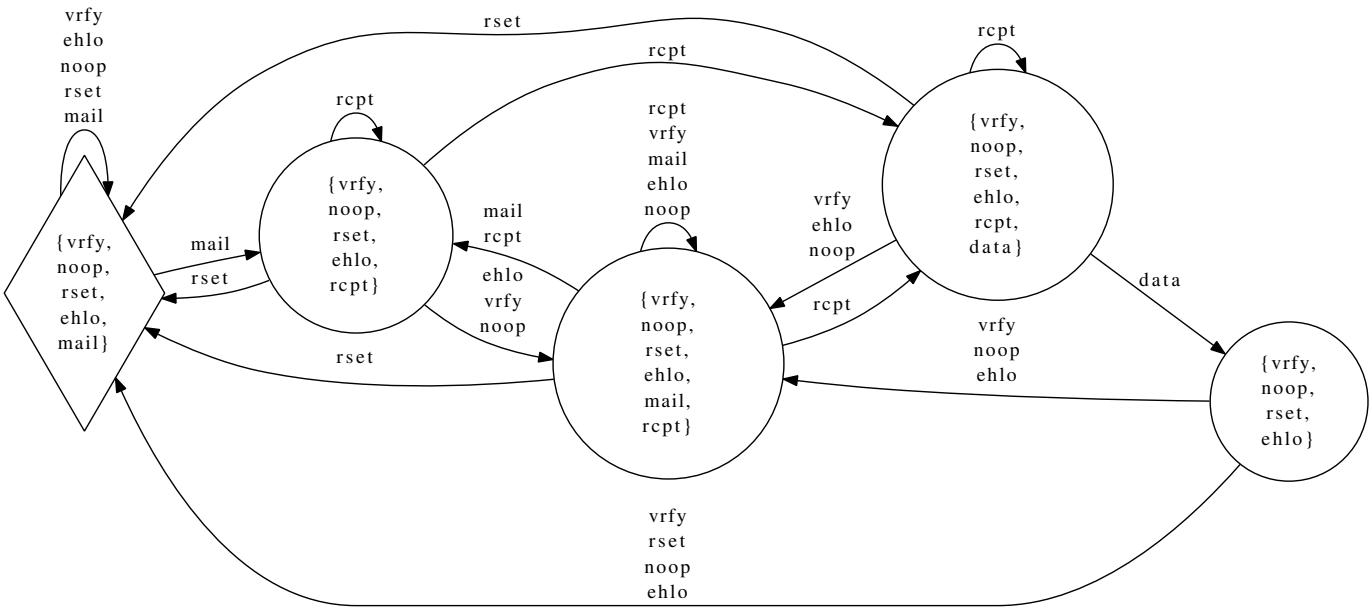

Figure 6.19: EPA for SMTP protocol server class

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.

6.3.7. SMTP Client

The `SMTPProtocol` class is a Java implementation of an SMTP protocol client extracted from the RISTRETTO Java mail client[4].

Using this class we created an action system. 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. We obtained the EPA in Figure 6.20.

When compared to the manually-generated model in [DKM+10] (see Figure 6.21) 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`).

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 [KFR+95]).

[4] http://ostatic.com/ristretto

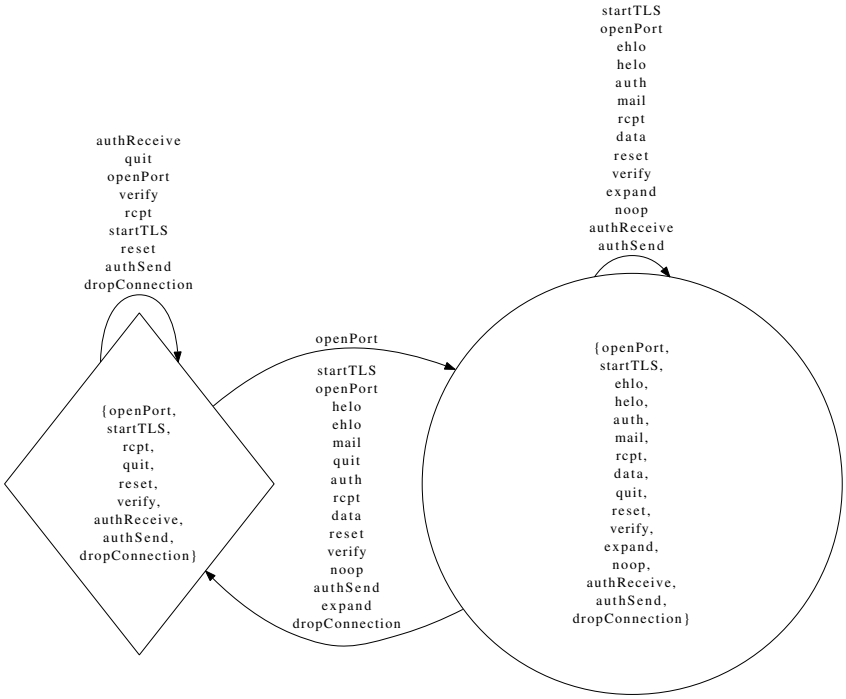

Figure 6.20: EPA for SMTP protocol client class

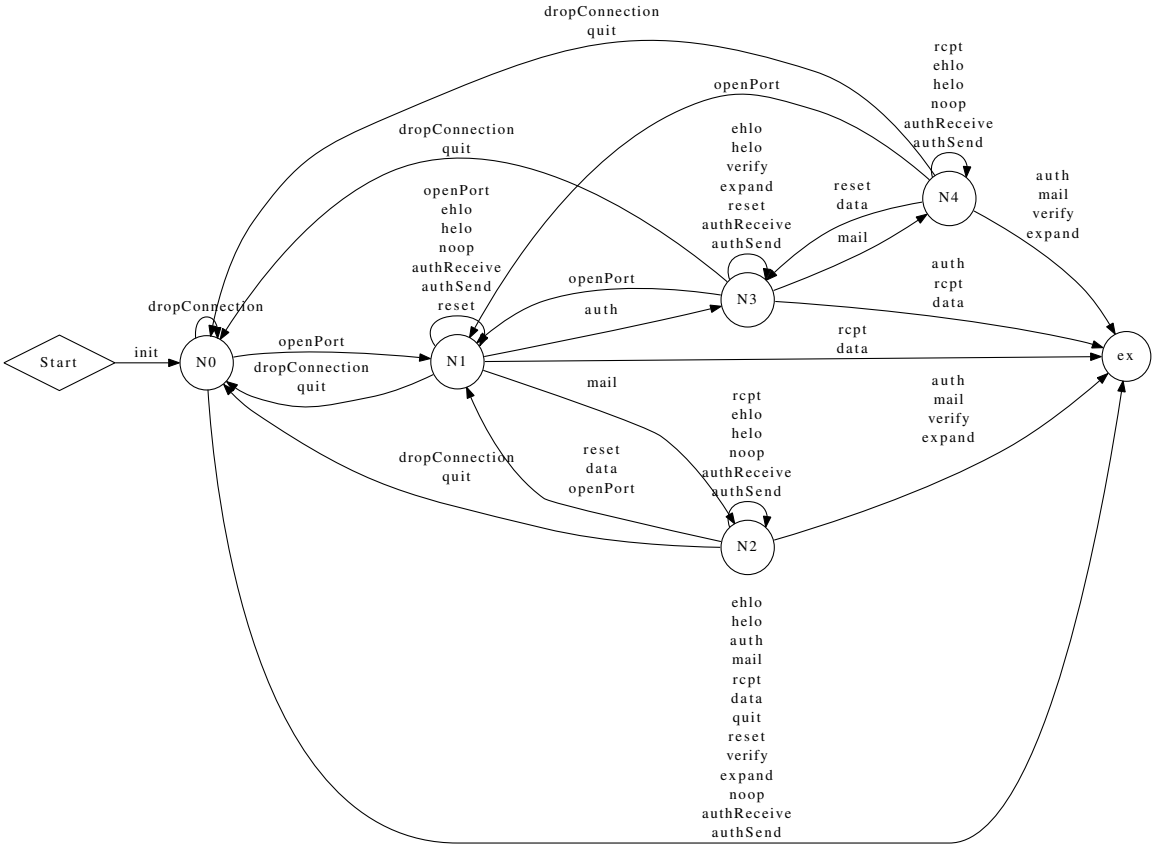

Figure 6.21: Manually generated model for SMTP protocol client class (extracted from [DKM+10])

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.

6.4. EPA Validation Guidelines

Based on our experience in various real-life software such as the ones introduced so far, 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 relate the different abstract states to 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 represent particular situations of the system under analysis. As an example of this, identifying empty, half-full and full abstract states in the circular buffer example is straightforward (see Figure 2.3).

  When it is not possible or not easy to associate a particular state 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) and that the requires clauses of enabled actions or the system invariant were (incorrectly) too weak. This is the case in Figure 2.2, in which the invariant was missing the requirement $wp \neq rp$.

- **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 of the bug in the ATM of [WSCF00] from the EPA in Figure 6.4.

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 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, presented in Section 6.2.4.

- **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 Section 6.2.3 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 6.16 presents a sink state {`close`} which merits attention.

- **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 Section 6.2.4 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 {`closeTCPConnection`, `gssInitSec`} abstract state in Figure 6.7 presents a highly non-deterministic `gssInitSec` 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 `vrfy`—`noop` in the SMTP protocol server EPA depicted in Figure 6.19.

Some of the heuristics presented in this section can be easily automated. In fact, our CONTRACTOR tool supports some of them.

CHAPTER 7

Studying EPAs Expressiveness and Understandability

We are interested in inferring models that can be directly consumed by developers. Such models face the challenge to remain simple enough to be understandable, yet interesting enough to convey useful information. In other words, we believe that in order to be helpful as a reviewing tool, coarse-grained abstractions have to be *expressive* enough to capture interesting behaviour, but still remain *understandable* to developers.

However, to the best of our knowledge, the expressiveness of models aimed to be consumed by developers has not yet been formally studied. Furthermore, evidence of developer understanding of such models is often missing or anecdotal in literature [CZvD+09, TTDBS07], with no statistical relevance. Moreover, few user studies have been conducted, mostly with negative results. For instance, in the context of models in the form of likely invariants, a recent user study [SHKR12] has shown that developers struggle to correctly understand them.

In this chapter we study these both expressiveness and developer understandability, in the context of EPAs. More concretely, we analyse EPA expressiveness by conducting an evaluation of how sensitive they are to the presence of defects in the source code of the API implementation. Regarding understandability, we conducted 3 user studies where developers were presented with an EPA that was either obtained from the original API implementation, or from a defective version of it. Developers were asked to identify whether these EPAs matched or not their mental model of the expected API behaviour. The overall goal of this study was twofold: to determine user detection effectiveness for EPAs that were generated from defective versions of the source code; and to understand what factors lead to successful or unsuccessful detection of these EPAs.

Our key results are twofold. First, despite their compactness, EPAs are still expressive enough to be affected by most of the defects we studied. Furthermore, EPAs led to the successful detection of defects that were subtle enough to be undetected by extensive testing. Second, users are quite effective at identifying whether EPAs were produced from the original source code or from defective versions of it. While EPAs from defective versions of the source code were correctly classified more often, we did not find statistically significant differences with respect to user effectiveness at classifying EPAs from the original source code. Finally, we identified two key factors that led to correct classification. We found that *i)* the biggest the change the EPA suffers from a defect (measured in number of transitions that change with

respect to the original version), the more likely users are to classify it correctly. And *ii)* effectiveness at classifying EPAs for the original source code had a positive correlation with participants' active dedication to programming. We believe that the obtained results provide evidence that coarse-grained abstractions are valuable aids during key software development tasks such as validation.

The rest of this chapter is organised as follows. In Section 7.1 we present our research questions. Section 7.2 focuses on analysing the impact of source code defects in the EPAs. In Section 7.3 we analyse whether developers are proficient at identifying the affected EPAs. Section 7.4 presents a discussion of the obtained results and their implications. We conclude with some final words in Section 7.5.

The raw data analysed throughout this chapter is available online at:

http://userstudies.lafhis-server.exp.dc.uba.ar/userstudy/analysis

7.1. Research Questions

In previous chapters we showed how, in order to remain understandable, EPAs abstract away elements of the original implementation. This expressiveness/understandability trade-off is a balancing act. We believe that EPAs provide a good balance between the two.

In this section we aim to provide empirical evidence to support our claim. In other words, we want to show that *i)* EPAs are *expressive* enough to provide valuable assistance throughout an API validation process, and *ii)* developers do *understand* EPAs.

We therefore state two research questions, which complement our first one (introduced in Chapter 6).

> **R.Q. 3:** Are EPAs expressive enough to be affected by defects in an API's source code?

> **R.Q. 4:** Given an EPA that was affected by a defect in an API's source code, can a reviewer understand it and realize that it is not in line with the expected behaviour?

First, in R.Q. 3 we state that we need EPAs to be expressive enough to react to changes in the source code of an API implementation. In other words, given two (semantically) different API implementations it is desirable that the EPAs for these two implementations differ. Otherwise, the reviewer will not be getting enough information that could assist her in classifying a correct implementation from a bogus one.

But showing that EPAs are affected by defects in the source code is not enough. Since we are dealing with a human-driven validation process, in R.Q. 4 we ask whether a developer can actually understand the EPA and notice that the resulting EPA is not in line with respect to her mental model.

We used four case examples when answering our research questions: `PipedOutputStream`, `Signature`, `ListItr` and `Socket`. All of these classes were already discussed in Chapter 6.

7.2. Analysing Code Defects Impact on EPAs

In this section we answer our R.Q. 3, which deals with how sensitive EPAs are with respect to defects in the original source code.

7.2.1. Experimental Setup

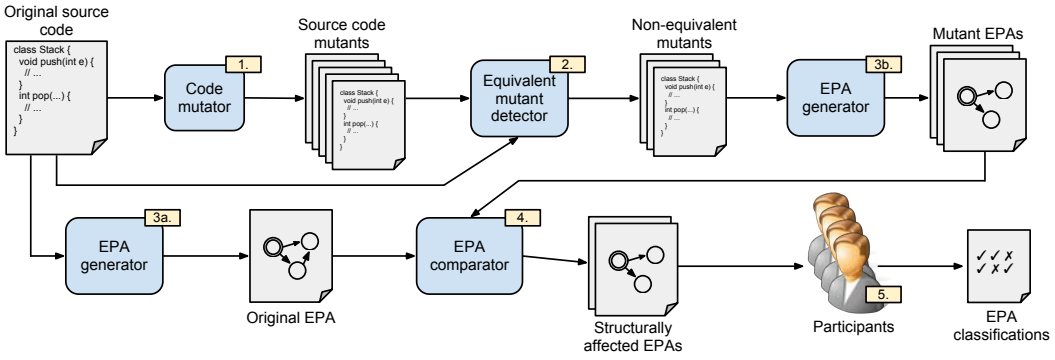

Figure 7.1: Experimental setup

Figure 7.1 presents the experimental setup that we used for answering our research questions. Here we focus on steps *1–4*, aimed at answering R.Q. 3.

In the absence of a large representative set of defects for the Java classes under analysis, we used mutant versions of the source code as a proxy. Given an API implementation, in step *1.* a *code mutator* is use to obtain a large number of API implementation mutants.

In order to discard mutants that do not produce defects, in step *2.* mutants are classified as either semantically equivalent to the original or not. Deciding if two programs are semantically equivalent is non decidable in general so we took a conservative approach based on regression testing.

Then, CONTRACTOR is used to generate EPAs for the original implementation (step *3a.*) and each of the non-equivalent implementation mutants (step *3b.*).

In step *4.* each mutant EPA is structurally compared with the original one in order to check whether they differ or not.

We finally obtained a profile of how many non-equivalent implementation mutants actually produced changes in the EPA and how big these changes are in terms of number of abstract transitions that were added/removed.

In the rest of this section we provide detailed explanations for each of these 4 steps, as well as the obtained results.

7.2.2. Mutant Generation

In order to obtain mutant versions for each of the case examples we used an automated tool called MILU [JH08]. MILU takes a C program and applies a number of standard mutation operators such as changing unary or binary operations, modifying constants or removing statements. It is worth mentioning that MILU generates bogus mutants that do not even compile, and these were discarded.

7.2.3. Discarding Semantically Equivalent Mutants

We used testing in order to conservatively discard semantically equivalent mutants. MILU not only generates mutants, it can also be used as a test harness to compare a test-suite's results on the original version and on the generated mutants.

If a mutant differs in at least one test-case then it is safe to assume that it is not semantically equivalent. On the other hand, if it yields the same results for all the test-cases it may be either equivalent to the original or it can also be the case that the test-suite is not powerful enough to distinguish them.

In order to generate very large test-suites (i.e., that would run for days) for each case example we constructed an ad-hoc random test-case generator. This generator takes a *description* of the case example. This case example description comprises the list of public methods that the test-suite should consider, divided in two groups: A method is an *action* if it changes the class observable state; a method is an *observer* if it returns a (usually read-only) fragment of the class internal state without altering it.

For the case of methods that take parameters the description provides a pool of concrete arguments that can be used to instantiate them. In our setting, this sufficed, since all of the methods in the APIs that we analysed have basic data types as parameters.

The random test-suite generator then proceeds as follows:

1. Create a set of traces T. Each trace $t_i \in T$ is essentially a sequence of n_i action methods $a_1, \ldots, a_{n_i}$. For each action method that takes parameters, the generator randomly chooses from the pool of concrete arguments.

2. For each trace $t_i \in T$ and each observer o_j construct a test-case that first sequentially invokes each methods in t_i and then returns the result of invoking o_j.

The number of traces is configurable, and regarding their length the user can provide a mean value n and the lengths for the traces are generated following an $exp(\frac{1}{n})$ distribution.

A random generator like the one we just described has a potential problem that if the case example protocol is non trivial, then many of the traces in T might be unfeasible. This causes the test-suite to take longer to run, without adding any valuable information. In order to solve this problem we first check that each trace is feasible in the original class before adding it to T. Since we also want to test whether the mutants add new legal behavior, there is one caveat. If a trace t is unfeasible but has a feasible prefix, we add this prefix together with the first illegal action to the set of traces T. In other words, we do want to test what happens in the boundary of the legal behaviour.

7.2.4. Generating Mutant EPAs

Generating EPAs for each of the mutants is a straightforward yet lengthy task. Each EPA is constructed in a time that ranges from a few seconds to around 20 minutes. Given that we work with hundreds of mutants, we applied a distributed approach in which several desktop computers were assigned a partition of the mutants to work with and generate their EPAs.

7.2.5. Structurally Comparing Mutant EPAs

Comparing graphs in general is a challenging problem. In the particular case of EPAs, the level of abstraction is fixed and the states are consistently labeled. Therefore checking for isomorphisms was trivial. If an EPA produced from a mutant is non isomorphic to the original, we say it is an *structurally affected* (or simply *affected*) EPA.

7.2.6. Threats to Validity

We use non-equivalent mutants as a proxy for defects, thus ensuring that every considered mutant is indeed a defective version of the source code. Even when

	Mutants			Test-cases			
Case example	Total	Compilable	Killed	Number	Mean length	Running time	Stmt. coverage
`ListItr`	2419	1846 (76.31%)	1308 (70.86%)	15582	15	1d 23h 15m	98.55%
`Signature`	381	259 (67.98%)	253 (97.68%)	18302	20	3d 15h 43m	100%
`PipedOutputStream`	518	483 (93.24%)	292 (60.46%)	22253	15	16d 8h 53m	93.02%
`Socket`	1069	1028 (96.16%)	539 (52.43%)	10240	20	1d 6h 53m	89.9%

Table 7.1: Overview of case examples and results for R.Q. 3

	Total	Killed mutants effect on EPA			
Case example	killed mutants	Adds transitions	Removes transitions	Adds and removes transitions	No effect
`ListItr`	1308	284 (21.71%)	480 (36.70%)	238 (18.20%)	306 (23.39%)
`Signature`	253	58 (22.92%)	52 (20.55%)	62 (24.51%)	81 (32.02%)
`PipedOutputStream`	292	103 (35.27%)	14 (4.79%)	34 (11.64%)	141 (48.29%)
`Socket`	539	112 (20.78%)	27 (5.01%)	129 (23.93%)	271 (50.28%)
	Total	Surviving mutants effect on EPA			
Case example	surviving mutants	Adds transitions	Removes transitions	Adds and removes transitions	No effect
`ListItr`	538	39 (7.25%)	242 (44.98%)	15 (2.79%)	242 (44.98%)
`Signature`	6	0 (0.00%)	1 (16.67%)	3 (50.00%)	2 (33.33%)
`PipedOutputStream`	191	65 (34.03%)	4 (2.09%)	0 (0.00%)	122 (63.87%)
`Socket`	489	83 (16.97%)	25 (5.11%)	20 (4.09%)	361 (73.82%)

Table 7.2: Mutant impact on the EPAs

mutants are generally used in literature [JH11], we might be missing other interesting potential bugs due to limitations of the mutant generator.

When deciding if mutants are semantically equivalent, we use a conservative test-based approach to solve an undecidable problem. Furthermore, the test-case generation strategy that we followed is a fairly straightforward one. We could have opted for more sophisticated techniques such as concolic testing (e.g., [SA06]).

However, as we show next, a manual inspection of the surviving mutants shows that they are intrinsically hard to identify by random test-case generation. In other words, we believe that we are close to the limits of semantically-equivalent mutant detection offered by this strategy.

7.2.7. Results

Table 7.1 presents the *total* number of mutants generated by Milu for each of the case examples. The number varies significantly from a few hundreds (as in the case of `Signature`) to a few thousands (in the case of `ListItr`). This depends on the amount of statements that Milu can change, which roughly increases with the size of the class implementation.

This same table also presents the total number of *compilable* mutants, which varies from almost all of the total mutants (as in `Socket`) to roughly two thirds (as in `Signature`).

The number of *killed* mutants is obtained by executing the test-suite described in the rightmost portion of Table 7.1. These test-suites were constructed using the approach described in Section 7.2.3. The proportion of non-equivalent compilable mutants varies between 52–97%.

Table 7.2 presents the proportion of compilable mutants that either add or remove transitions to the EPA with respect to the original EPA.

When considering non-equivalent mutants (upper half of the table), a rather large proportion (23-50%) of EPAs remain unaffected. It is understandable, however, since EPAs focus on the protocol aspect of the implementation, and may remain unaffected by mutants that alter the way data is handled (and not the ordering of operations). By being agnostic to the kind of mutations that Milu seeded, we can certainly avoid bias.

Case example	Surviving mutants adding transitions	Manual classification		
		Equivalent mutant	Non-equivalent mutant	Depends on memory model
`ListItr`	39	2 (5.13%)	7 (17.94%)	30 (76.92%)
`PipedOutputStream`	65	5 (7.69%)	14 (21.54%)	46 (70.77%)
`Socket`	83	5 (6.02%)	27 (32.53%)	51 (61.44%)

Table 7.3: Manual analysis of surviving mutants that add transitions

A large number (50–75%) of the killed mutants produce structurally affected EPAs.

Regarding the kind of impact, it varies significantly from one case example to another. For instance, in the `ListItr` case example almost 55% of the killed mutants produce EPAs that have missing transitions with respect to the original one. On the other hand, when considering the `PipedOutputStream` class, only 16% of the mutants result in transitions being removed from the resulting EPA.

In the lower half of the table we have the surviving mutants. These are mutants that could not be proven non-equivalent after (hours and even days of) random testing with good statement coverage. A priori, we would expect these mutants to be actually equivalent, and should therefore see a low number of EPAs being affected.

However, despite being coarse-grained, EPAs can also provide solid evidence that a mutation is indeed non-equivalent. More concretely, if a sequence of actions is not exhibited by an EPA, this indicates that the class does not admit that invocation sequence. Furthermore, EPAs are minimal in the sense that, if there are no uncertainties coming from the decision engine (either SMT or software model-checker) and the specified invariant is tight, they do not add a transition unless it is strictly required to capture a legal invocation sequence.

Combining these two observations, if a mutant yields an EPA that is missing at least one transition with respect to the original EPA, this means that there is at least one invocation sequence in the original class that is not permitted in the mutant. This implies that the mutant is non-equivalent.

Following this result, we now analyse the lower half of Table 7.2. Mutants that either remove transitions only, or that both remove and add transitions are definitely non-equivalent, despite having survived the test-suite. Adding up the number of EPAs that remove transitions and the ones that add and remove transitions, we can see how in the first two rows, the EPAs help us identify that 48–67% of the surviving mutants are indeed non-equivalent. This result is remarkable considering that these are mutants that survived a random testing approach for several hours (and even days). In the last two case examples the EPAs only identify between 2–9% of new non-equivalent mutants.

In 2 examples, 48–67% of the surviving mutants remove transitions from EPAs, which proves that they are non-equivalent and therefore were missed by the test-suite.

On the other hand, between 7–34% of the surviving mutants yield EPAs that only add transitions. This can indicate that either the mutant is indeed equivalent but the EPA coarse-grained nature is causing new transitions to appear, or that the new transitions are evidence of the non-equivalence of the mutant. Since the EPAs provide a superset of the behaviour, deciding which of the two cases requires manual inspection for each of the mutants. Table 7.3 summarizes what we discovered from manually inspecting each of these mutants.

Case example	Non-equivalent mutants	Non-equivalent mutants detected By the test-suite	By EPAs
`ListItr`	1602	1308 (81.65%)	1012 (63.17%)
`Signature`	257	253 (98.44%)	118 (45.91%)
`PipedOutputStream`	356	292 (82.02%)	112 (31.46%)
`Socket`	662	539 (81.42)	279 (42.15%)

Table 7.4: Mutant detection capabilities of EPAs and test-suite

As we observe, only a few (5–7%) of the surviving mutants that add transitions are semantically equivalent to the original class. An important number (17–32%) of mutants are non-equivalent, which indicates that the manifestation in the EPA is not a false positive. The remaining vast majority (61–77%) of the mutants is semantically equivalent modulo uncontrollable decisions such as to which (arbitrary) value the memory locations are initialized[1].

Only 5–8% of the equivalent mutants add transitions to the EPA, which indicates a very low rate of false positives.

Finally, Table 7.4 compares how many mutants are killed by the test-suites with how many mutants are detected by the EPAs. The total number of mutants known to be non-equivalent contains: *a)* mutants killed by at least a test-case, *b)* mutants that remove at least one transition from the EPA, and *c)* manually classified mutants. As we observe, the test-suites consistently detect many more mutants than the EPAs. This is reasonable, considering that EPAs are targeted at capturing the usage protocol, while test-suites can detect other mutations which only affect the internal state of the class. However, given that EPAs intentionally coarse-grained and are aimed at manual inspection, detecting roughly 50% the amount of killed mutants is not a bad performance.

7.3. Analysing User Understanding of EPAs

As we mentioned before, we are not only interested in observing how defects in the source code alter the EPAs, but also on discovering how developers understand this phenomenon.

Our R.Q. 4 refers to the user's ability to notice that structurally affected EPAs are not well aligned with the expected behaviour for the class. We answer this question by means of a series of 3 user studies, presented in this section.

7.3.1. Experimental Setup

Remember, from step *4.* in Figure 7.1 we obtained a set of structurally affected EPAs. We filtered out the EPAs that reported invariant violations (via the experimental CONTRACTOR feature discussed in Section 4.4.2) and chose 99 non-equivalent mutants that produced an affected EPA for each of the case examples, together with the original EPA for each case example. We took all these EPAs and presented them in random order to various participants. In step *5.* each participant was asked to classify each EPA as either "original" or "affected". Two potential types of errors exist: marking an original EPA as affected and marking an affected EPA as original.

[1]Remember that we are dealing with C versions of the classes.

7.3.2. Experiment Procedure

We performed the user studies during the anual winter summer school at the Universidad de Buenos Aires. This is a week in which a series of advanced courses are organized for local students and practitioners and also bringing people from other regions of Argentina and neighboring countries. In order not to clash with the actual winter school lectures, we performed each user study at a different time: one in the morning, another one in the afternoon and a third one in the evening. The software we used to run the experiments, as well as the materials and the procedure was exactly the same in the three shifts.

All of our controlled experiments began with the administration of a background questionnaire gathering information about participants' experience with behaviour models, industry experience, academic background, and so forth. We then offered a tutorial presentation *(i)* explaining the general concepts behind EPAs; *(ii)* introducing EPAs for small code examples like the ones in Chapter 2; *(iii)* explaining what it means for an EPA to be *affected* by a change in the source code and giving examples of affected EPAs; *(iv)* describing the tasks to be done and explaining the experiment front-end they had to use; and *(v)* providing example tasks for the participants.

Following the tutorial presentation, each participant was assigned a series of tasks. Each task consisted of classifying either the original EPA or one of the 99 mutant EPAs from one of the case examples we considered.

In order to familiarize themselves with the case examples under analysis, participants had access to reference materials. In particular, we provided access to the official Javadoc documentation and a read-only view of the source code. It is important to remark that participants were offered the original source code, regardless of whether the EPA they had to classify was from a mutant or not.

Participants were encouraged to classify the EPA and provide an optional free text explanation, but were allowed to leave it unclassified, for reasons of time or uncertainty. These unclassified EPAs were categorized as "unknown".

Participants were alloted up to 12 minutes to complete each task. This time limit was determined after test-driving the experiment with 6 beta-testers while imposing no time restriction and observing that all their tasks were completed in less than 9 minutes. The results for the experiment test-drive were discarded, but provided valuable feedback in refining the experiment infrastructure.

Participants were given up to 90 minutes to complete as many tasks as they could. In order to keep a high quality of participants responses, they were offered the possibility to quit the experiment early if they wanted. Actually, after 1 hour, more than two thirds of the participants had already left. We believe that keeping them past their will would have resulted in a lower quality of responses (e.g. participants may have started to randomly pick answers). However, as an incentive to keep participants engaged, we raffled prizes, weighing the length of their participation and the quality of their responses.

Once the 90 minutes were out or the participant decided to leave, exit surveys were administered, gathering information concerning participants' confidence in their classifications, their opinion of usefulness of the EPAs, and so forth.

7.3.3. Threats to Validity

External: In this study, all the participants were volunteers attending the winter courses at the university. However, most of them were professional software developers aged 20–46 with various degrees of experience. While 45% of participants never worked in software-related positions, 95% of all participants claimed to

have at least 1 year experience with Java. Furthermore, 35% of the participants had at least 3 years of industry experience in software-related positions.

Even when our case examples are well-known Java classes, they are usually used as black-box components and their source code is usually unfamiliar to Java developers. In practice, however, developers are often expected to maintain code they are unfamiliar with, since it is not theirs.

Furthermore, in our study, participants had limited time to complete the tasks. However, when beta-testing our experiment we observed that all the tasks were completed in under 9 minutes each; in the experiments we assigned one third more. In any case, in real-life situations developers are typically time limited due to project constraints. Chapter 6 presents a complementary study, in which a few experts were given unlimited time to use EPAs to validate the behaviour of the same case examples, among others.

Each of the case examples involves Java classes with varying degrees of difficulty in terms of API state-space or lines of code. It seems likely that other APIs that developers may use should have a comparable complexity to one of the case examples that we used in our experiments.

Internal: Our measurement of user effectiveness is obtained by comparing participants' EPA classifications against our own classifications. Our experimental setup guarantees that our classification is correct by construction, so we can therefore rely in our own classification as being 100% correct.

Construct: We measure user effectiveness at classifying EPAs as the proportion of correct answers over the set of EPAs that they were assigned to classify. Each participant classified around 25 EPAs chosen at random from a set of 400. Instead, we could have pre-selected a random set of EPAs and have all the users classified those. This would have resulted in a more consistent efficiency metric along users, but over a less representative set of EPAs.

Conclusion: We have conducted 3 case studies with 20 participants, which is a relatively small number of users. However, as we show next, it is sufficient to demonstrate statistically significant trends for some of our research questions. All statistical methods we employ are non-parametric [KV07], thus conclusions we make rely on few assumptions.

7.3.4. Research Question Refinement

In order to address R.Q. 4, we refine it as follows.

- **R.Q. 4A**: Were user classifications honest or did they follow a random pattern or similar strategy?
- **R.Q. 4B**: Are users more effective at classifying original EPAs than classifying affected EPAs?
- **R.Q. 4C**: How is user classification effectiveness influenced by class complexity/size?
- **R.Q. 4D**: In the context of affected EPAs, how is user classification effectiveness influenced by the size of the change with respect to the original EPA?
- **R.Q. 4E**: How is user classification effectiveness influenced by the participants background and experience?

Case example	Classification of original EPAs Total	Correct	Incorrect
`ListItr`	11	3 (27.27%)	8 (72.73%)
`Signature`	17	13 (76.47%)	4 (23.53%)
`PipedOutputStream`	15	7 (46.67%)	8 (53.33%)
`Socket`	15	11 (73.33%)	4 (26.67%)

Case example	Proportion of correctly classified affected EPAs Min	Mean	Median	Max	Std. dev.
`ListItr`	0.00%	87.00%	100.00%	100.00%	30
`Signature`	25.00%	77.00%	93.00%	100.00%	28
`PipedOutputStream`	0.00%	81.00%	96.00%	100.00%	28
`Socket`	25.00%	74.00%	75.00%	100.00%	24

Table 7.5: Proportion of correctly classified EPAs

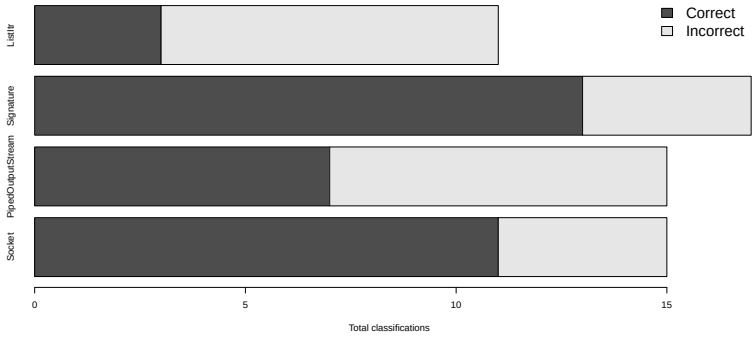

Figure 7.2: User effectiveness classifying original EPAs

7.3.5. Results

For each user's assessment of an EPA, the EPA is either original or affected, and the user may judge it to be either original, affected, or unknown. Thus for each assessment of an EPA, six basic results can occur. Only two categories represent correct judgements by the users: classifying an original EPA as original, and classifying an affected EPA as such. In our analyses, we do not consider unknown classifications. The average number of unknown responses by participant ranges between 0.1–0.7 depending on the case example.

In Table 7.5 we present user classification effectiveness. In the case of original EPAs, we present the number of correct and incorrect classifications (at most one classification per participant). For affected EPAs, we analyse the percentage of them correctly classified per participant, including the standard deviation (in percentage points).

A graphical representation of the information in Table 7.5 can be found in Figures 7.2 and 7.3. In the first, we depict the proportion of correct classifications for each original EPA. The latter presents, for each of the case examples, the proportion of affected EPAs that each participant correctly classified, in the form of a box plot.

With respect to the number of classifications by each participant the average number of (non-unknown) completed tasks per participant is roughly 5 per case example with a standard deviation between 4.1–4.86.

In the remainder of this section we address each of the refined research questions presented in Section 7.3.4.

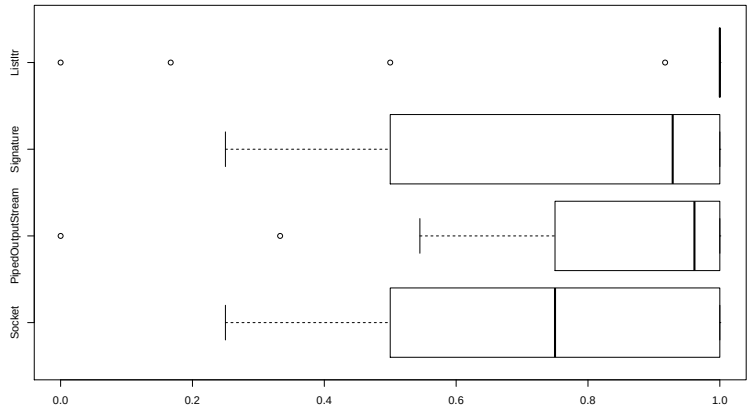

Figure 7.3: User effectiveness classifying affected EPAs

Case example	Normal fit with $\mu = 0.5$ and $\sigma = 0.5$ Affected EPA effectiveness p-value	Original EPA effectiveness p-value
`PipedOutputStream`	**0.0045**	**0.0242**
`Signature`	**0.0292**	**0.0004**
`Socket`	**0.0124**	**0.0022**
`ListItr`	**<0.0001**	**0.0308**
Overall	**<0.0001**	0.6284

Table 7.6: p-values for normal fit of H1

R.Q. 4A: Disproving Randomness of Classifications

Before diving into the analysis of user effectiveness, we want to make sure that user classifications are honest and not product of randomness (or other ad-hoc strategies).

In particular, we wanted to test whether user's classifications outperform an hypothetical participant that classifies randomly following a Bernoulli distribution with parameter 0.5. If that were the case, then the effectiveness (percentage of correct answers) should follow a normal distribution with $\mu = 0.5$ and $\sigma = \sqrt{\mu(1-\mu)} = 0.5$. To test whether this is the case, we proposed the following hypothesis and null hypothesis:

H1: User classifications are not randomly chosen.

H1$_0$: User classifications are randomly chosen and therefore user effectiveness follows a normal distribution with $\mu = 0.5$ and $\sigma = 0.5$.

Table 7.6 presents the p-values obtained after fitting the user classification effectiveness data to a normal distribution with $\mu = 0.5$ and $\sigma = 0.5$. The last row corresponds to the overall effectiveness for each user across case examples. As we observe, p-values are sufficiently low to reject the null hypothesis H1$_0$ with $\alpha = 0.05$ on most cases, which implies that participants did not resort to a random strategy. The exception is the user overall effectiveness when classifying original EPAs, but as Figure 7.2 suggests, this is just the result of averaging mixed results.

Furthermore, for each case example we prepared 100 classification tasks consisting of 99 affected EPAs plus the original EPA. Had a participant known about this proportion, a strategy of always classifying EPAs as affected would have led to a very high effectiveness.

In order to show that participants did not know the original/affected EPA proportion, we analysed the ratio of total EPAs classified as original for each participant. These ratios can be fitted with probability 0.58 to a normal distribution with $\mu = 0.268$ and $\sigma = 0.177$. While participants tended to classify EPAs as affected more often, the average original classification ratio is significantly larger than $\frac{1}{100}$, which is what an informed participant would have obtained.

Finally, we analysed whether there exists a learning effect on participants. That is, whether their effectiveness increases over time. In order to do so, we computed the overall effectiveness for each participant at two points: *i)* when it completes the task $n/2$ where n is the participant's total number of tasks, and *ii)* when it completes the task n. In other words, we measured the participant effectiveness both nearly in the middle of his participation and at the end of the experiment. If there was a learning effect, the population of second values should be significantly larger than the first. To test this hypothesis, we use a non-parametric test for evaluating whether two sets of numbers are drawn from the same population. In this case, each user provides two data points —effectiveness at mid-experiment and final effectiveness— and therefore the data for evaluating this hypothesis is paired, with 20 pairs, one for each participant. We thus applied the two-tailed Wilcoxon test and found no evidence that the median differences are different than zero (p-value = 0.3896). In other words, we found no evidence of a learning effect.

R.Q. 4B: User Effectiveness for Original and Affected EPAs

As Figure 7.3 shows, while original EPAs are often misclassifed by users (27-75%), affected EPAs are generally correctly classified (75–86%).

In some case examples, users seem to do better at classifying affected EPAs (e.g. for `ListItr` the difference is remarkable). To test this, we proposed the following hypotheses:

H2: Users are more effective at classifying affected EPAs than original EPAs.

$H2_0$: The percentage of correct classifications by users for affected and original EPAs are drawn from the same distribution.

As before, since we were dealing with paired data (effectiveness for original EPAs and affected EPAs for each participant), we applied the two-tailed Wilcoxon test for each case study.

In the `ListItr` case example we obtain a p-value of 0.0148, so we reject the null hypothesis at $\alpha = 0.05$. Given the means observed in Table 7.5, this implies that there is statistical evidence that users are more effective at classifying `ListItr` affected EPAs than the `ListItr` original EPA. When considering `PipedOutputStream`, `Signature` and `Socket`, the p-values are high enough (between 0.19 and greater than 0.99) to indicate that H2 is not supported.

Since there is evidence in favour of H2 in only one case example, we can not reject the null hypothesis.

R.Q. 4C: Impact of Program Complexity on User Effectiveness

In Table 7.5, we infer from the mean and median user effectiveness that there exist differences in user effectiveness between case examples. However, as indicated in Figure 7.3, the interquartile ranges are often quite large, usually larger than the differences between case example medians.

	Signature	Socket	ListItr
PipedOutputStream	0.8740	0.8628	0.2396
Signature	-	>0.9999	**0.0136**
Socket	-	-	0.1408

Table 7.7: p-values for Wilcoxon test of H3

	Correlation coeff.	p-value
Added transitions	-0.02	0.73
Removed transitions	0.09	0.17
Changed transitions	0.21	**0.0005**

Table 7.8: Affected transitions — effectiveness correlations

To further study the impact of program complexity, we again employed statistical hypothesis testing. Accordingly, we formulated the following hypothesis and null hypothesis:

H3: Users are more effective at classifying affected EPAs for smaller, less complex programs than for larger, more complex ones.

H3$_0$: The percentage of correct affected EPA classifications for two case examples of different size/complexity are drawn from the same distribution.

We evaluate this hypothesis for each pair of case examples. We again use a paired Wilcoxon test, resulting in the p-values shown in Table 7.7.

We can only reject the null hypothesis at $\alpha = 0.05$ level for the `Signature`-`ListItr` pair of case examples. As we observe from Table 7.5, the mean user effectiveness when dealing with affected for the `ListItr` case example is bigger than for the `Signature` case example. However, from Table 7.1 we observe that actually the `ListItr` case example is larger (in LOC) than the `Signature` case example, which contradicts H3. We thus conclude that H3 is not supported.

R.Q. 4D: Impact of EPA Change on User Effectiveness

Having found no evidence that class complexity affects user effectiveness, we now try to establish a different explanation. More concretely, it is intuitive to think that the biggest the change on the EPA that a mutation on the source code produces, the more likely it will be that a user notices something odd with the EPA.

Formally, it would seem reasonable to find a correlation between the number of transitions changed (that is, either added or removed) from an affected EPA and the user effectiveness on that EPA. A Spearman (non-parametric) correlation test yields the results in Table 7.8. The correlation coefficient ranges from -1 (full inverse correlation) to 1 (full direct correlation), and the p-value indicates its statistical significance.

While there is a statistically significant correlation between effectiveness and the number of changed transitions on an EPA, the correlation coefficient of 0.2 is considered indication of *weak* correlation.

R.Q. 4E: Impact of User Background on Effectiveness

Finally, in order to explain user effectiveness we analyse the participant's background trying to establish correlations with their effectiveness. It would seem natural to think that the more experienced a participant is, either academically or industrially, the better she should score when classifying EPAs.

	Effectiveness			
	Affected EPAs		Original EPAs	
	Corr. coeff.	p-value	Corr. coeff.	p-value
Hours/day	0.02	0.94	0.48	**0.04**
Years IT	0.33	0.15	0.23	0.34
Years Java	-0.17	0.47	-0.19	0.43
Education	0.02	0.95	0.23	0.34
Exp. models	-0.17	0.48	0.15	0.55

Table 7.9: Participant's background — effectiveness correlations

We analysed data collected during the pre-experiment survey, such as amount of hours worked per day, number of years of experience in IT-related jobs, number of years of Java experience, level of education (e.g., grad, undergrad) and whether they had experience with formal behaviour models.

The collected data is presented in Table 7.9. The only statistically significant correlation indicates that the more hours per day a participant works, the better he is at identifying original EPAs. The correlation coefficient is 0.48, which indicates a *moderate* correlation. When correlating user effectiveness on each individual case example with the same factors, no statistically significant correlations were consistently identified.

7.4. Discussion

Despite being a rather coarse-grained abstraction aimed at human inspection, EPAs are usually affected by the presence of defects in the API source code. Most of the non-equivalent mutants produced changes on their EPAs. Not only that, EPAs helped in the identification of several non-equivalent mutants that had survived after extensive testing.

Regarding affected EPAs, in the majority of cases participants were able to detect that they were not aligned with the expected behavior. Furthermore, we found no evidence that EPAs were randomly classified (or that participants followed an ad-hoc strategy), which leads us to believe that users consciously reviewed the EPAs and made careful classifications.

With respect to original EPAs, we found mixed results. While `Socket` and `Signature` exhibit an effectiveness similar to that of affected EPA classifications, in `PipedOutputStream` and `ListItr` average efficiency was below 50%. We believe, and our experience so far confirms, that `ListItr` offers the most complex usage protocol of all four classes. In a previous work [dCBGU11] an expert Java developer was asked to manually create a behaviour model for the class and the result had serious flaws. In fact, a couple user classifications for the original `ListItr` were accompanied with comments that entailed strong misconceptions about the class. With `PipedOutputStream` user misconceptions about the class are more evident: roughly half of the classifications have erroneous comments.

Explaining *why* users were effective classifying EPAs is a challenging problem. While it should be natural that the bigger the change on the EPA, the more likely it is to be detected by users, the obtained data only shows a weak correlation. Finally, we found a moderate correlation between effectiveness and current professional dedication to programming (in terms of hours/day). Other factors such as years of experience had no statistically relevant correlations with effectiveness.

7.5. Conclusions

We have analysed the expressiveness and understandability of an intentionally coarse-grained behaviour abstraction. We believe that the obtained results support the value that coarse-grained abstractions have in assisting developers in key tasks such as validation. A natural next step, in the case of EPAs, would be to explore how to carefully refine them without compromising understandability. While these abstractions are specially suited to work with classes featuring rich usage protocols, we envision they can be complemented with other types of abstractions when dealing with other kinds of programs.

Part IV

Discussion

CHAPTER 8

Related Work

In this section we compare our approach to the construction of behaviour models with other previously published techniques. Table 8.1 presents a summary comparison with some of the most prominent comparable approaches.

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 or pre/post-condition specification, 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.

8.1. Static Typestate Inference

Our technique is related to approaches that synthesize typestates [SY86, DF01, NGC05] or interfaces [AČMN05, GP09, HJM05] 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 [GP09, HJM05] 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 [AČMN05]. 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 [NGC05] 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

Technique	Input	Output	Construction	Purpose
[AČMN05]	API source code	Model that accepts subset of legal traces	Predicate abstraction, language learning	Verification of API client usage
[NGC05]	API source code	Model that accepts subset of legal traces	Predicate abstraction, abstract interpretation	Verification of API client usage
[HJM05]	API source code (finite internal state)	Model that accepts all legal traces	Predicate abstraction via software model checking	Verification of API client usage
[GP09]	Finite LTS	Model that accepts all legal traces	Software model checking, language learning	Compositional verification
[GS97]	Set of guarded-assignments, set of predicates	Model that accepts superset of legal traces	Predicate abstraction via assisted theorem proving	Verification of system properties
[GGSV02]	Abstract state machine	Underapproximation of the "true FSM"	Symbolic execution	Construction of test-suite
[LMS07]	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
[GS08]	API client traces	Model that accepts superset of observed traces	BDD-based mining algorithm	API specification recovery
[DLWZ06]	API client traces	Partition of concrete states according to observers output	Predicate abstraction over given traces	Construction of API test-suite
[GMM09]	API client traces	Model that accepts superset of given traces	Extrapolation via graph transformation rules	API specification recovery
[LMP08]	API client traces	Model that accepts superset of given traces, preserving data dependencies	k-tail, data invariant inference	Construction of API test-suite
[PG09]	API client traces	Model that accepts superset of observed traces	States model methods, edges model precedence frequency between methods	API specification recovery
[BBSE11]	API client traces	Model that accepts superset of observed traces	Extrapolation via transitive closure of temporal invariants	Human inspection
[DR09]	API client traces	Model that accepts superset of observed traces	Predicate abstraction using a set of built-in predicates	Human inspection

Table 8.1: Related work summary

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., [BA08]) 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 [BN11] the authors present a technique to automatically infer annotations for the client usages of the protocol.

8.2. Predicate Abstraction

Our work can be considered an instantiation of the predicate abstraction [Uri99] framework.

Within the area of predicate abstraction, a closely related technique is the construction of finite state machines from Z specifications (which include pre and postconditions) and Live Sequence Charts (LSCs) [SD06]. Although there are similarities with our work in how transitions are computed the key difference is in the predicates used for abstraction: In [SD06] predicates found in LSCs are used to construct the set of states, while pre and postconditions are used to construct transitions. We use pre and postconditions for constructing both the states and the transitions, thus leveraging the enabledness concept in order to generate models which are useful for validation. Other predicate abstraction approaches such as counterexample-guided abstraction refinement (e.g. [BHJM07]) sometimes need an initial model and a property from which then the iterative process is performed. In fact, we believe that EPAs may serve that first purpose.

8.3. Model Minimisation

Our work is also related to techniques that construct abstract state graphs from infinite state systems (e.g., [LY92, GS97, GGSV02]). 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 [GS97] to model the enabledness conditions of actions, the output would be too large for manual inspection (see [dCBGU12a] for further discussion). Notably, the setting in [GGSV02] 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.

In general, minimisation approaches do not deal with actions with parameters in the implicit expression of the transition system (our LTS may have infinitely many labels due to parameters). The exception seems to be [TY01] where the authors present a technique for obtaining an untimed abstraction of timed automata. In timed automata semantics, the LTS also features infinitely many time transitions, that is transitions labelled with a real number standing for time elapsed from the source state. The abstractions yield by that technique feature an abstract time transition when for every state represented by the source abstract state there exists an amount of time to elapse and thus change to a state which maps to the target of the abstract transition. That is, it works as an existential elimination of the parameter value. Similarly, our technique exhibits a transition at the abstract level if there may be at least one parameter value (and a concrete state) to jump to the

target abstract state. Unlike [TY01], we do not require every concrete state to be enabled to perform such a jump (i.e., we are not requiring pre-stability of the yielded abstraction).

8.4. Contract Exploration

Other specification validation techniques such as the ones presented in [GKM+08b, LB08, NFLTJ06] explore the state space of a given contract either symbolically or concretely but they do not intend to construct a complete finite abstraction of it. We believe the latter provides a global view that can aid the validation process in a complementary manner.

The ideas presented in [VvLMP04] are also aimed at validation of specifications by automatically constructing finite state machines from them. However, the construction does not involve further abstraction: the language used for the pre and postconditions requires bounding the number and values of propositions and predicates.

8.5. Model Synthesis from Requirements

Techniques that construct FSMs from declarative requirements specifications [LKMU08] have been proposed as a means to facilitate analysis of such specifications and to support the transition to more design oriented modelling techniques. A particular instance of these approaches is the construction of FSMs from pre/post condition specifications. This approach differs from ours in that of their pre/post condition specification language is propositional logic, the concrete state space is therefore finite modulo bisimulation and that the resulting FSM has the same level of abstraction as the specification.

8.6. Testing-related Approaches

A level of abstraction somewhat related to that of enabledness has been used in [LMS07]. 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 [LMS07] 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 [GKM+08a] 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.

8.7. Model Mining

Our approach relates to the mining of temporal specifications (e.g., [DLWZ06, GMM09, GS08, LMP08, DKM+10, BBSE11, PG09]), 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 overapproximations of the artefact under analysis behaviour. On the other hand, our technique *statically* yields a model that is an abstraction of the program's source code, considering all possible paths.

The main difference with [GS08] 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 [DLWZ06] 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 [GMM09], 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 [LMP08], 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 [LMP08] (with $k = 1$) of the infinite set of traces for a given program.

Finally, [PG09] 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.

8.8. 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 [BBSE11] 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 [BBSE11] 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 [DR09]. 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.

8.9. User-studies On Understanding of Inferred Models

While model inference is an active field of research, empirical evidence of how users understand and interact with the synthesised models is often missing or anecdotal [CZvD+09, TTDBS07, PGKG08]. However, in the last couple of years there have been some interesting controlled experiments in the area (e.g. [SHKR12]), and the tendency is gaining momentum.

The technique presented in [BBSE11], already discussed in this chapter, presents a model mining technique based in leveraging existing logging infrastructure. The authors claim that these models are suitable for human inspection and provide evidence from a case study involving a single developer as well as a user study involving 45 students divided in groups of 2–4. While authors found evidence of the usefulness of the mined models, their interpretation of the user study results is mostly qualitative, omitting statical analyses.

Another user study is presented in [SHKR12], where the authors analyse developers' ability to classify likely invariants produced by DAIKON [EPG+07] as either correct or incorrect. Similarly to our study, participants had to determine whether a behaviour abstraction is well aligned with their understanding of the original source code. In their setting, the initial classification of invariants involves human partici-

pation and is therefore not 100% accurate. In our setting we know by construction which EPAs correspond to the original program and which ones to defective versions of it. Another qualitative study of inferred likely invariants is presented in [PCM08].

More recently, in the context of program verification, [DDA12] explores whether a static analysis technique can help users to manually determine if the failure to verify a program was due to limitations of the engine or due to an actual bug. They present a user-study which provides statically-significant evidence and show that the hints given by their tool dramatically increased user accuracy. The aim of their study is similar in the sense they show that the outcome of a program analysis technique can facilitate manual validation.

CHAPTER 9

Conclusion

In the first part of this work we introduced and studied enabledness-preserving abstractions (or EPAs), a coarse-grained behaviour abstraction for an API usage protocol which is aimed at human inspection. We presented algorithms that can statically and automatically construct EPAs either from an API specification or an API implementation. These algorithms were implemented into an open-source tool called Contractor, which includes a number of features designed to assist developers using EPAs for validation tasks.

The second part of this document presented evidence in favor of our claim that EPAs are a valuable companion for developers. More concretely, we reported the result of a series of case studies where experts analised EPAs for industrial-strength APIs uncovering interesting findings on the way. As a consequence, we crafted a list of validation guidelines that can help reviewers identify suspicious EPA elements. We complemented these case studies by exploring how sensitive EPAs are: a key question when dealing with a coarse-grained abstraction. The last part of our validation was via three controlled experiments where developers were asked to decide whether an EPA presented suspicious elements or not.

On the last part of this work we explored related approaches. To the best of our knowledge our approach is the first to statically and automatically construct overapproximated models suitable for human inspection.

9.1. Future Work

We believe that there are various items that would improve our approach.

First, we plan to improve the tool support for our approach to validation. Automatic extraction of the requires clauses and guessing a candidate invariant would make it much easier for developers generating EPAs. Automatic splitting of requires clauses is also of interest, specially since our current approach deals automatically with a limited set of requires clause patterns.

Automatically suggesting interesting EPA refinements to the developer appears promising, specially in very large EPAs. Refining the model in real-time while the developer explores it could create a "zooming" effect that could help him to get a better understanding. Our tool can also be enhanced by automating as many items on our validation guidelines as possible.

Exploring other EPA construction mechanisms also appears as an interesting

research path. We have already developed some preliminary results in generating EPAs from .NET code using a verification engine in [ZBdC+11]. That same work explores the possibility of using EPAs to verify client code, which is not their intended use, but still offers promising possibilities.

Finally, we believe that we can augment EPAs to include elements from other types of models aimed to be consumed by humans, such as likely invariants or usage scenarios.

9.2. Outlook

Overall, we believe, and our experience confirms, that EPAs provide an interesting trade-off between expressiveness and understandability. In other words, EPAs are sensitive enough to be altered by most of the seeded defects that we considered, but still simple enough for most of the developers to actually understand these changes. Our experience with the case studies confirms our claim that EPAs can indeed play an important role in the validation of software artefacts with non-trivial usage protocols.

More importantly, we believe that this work opens the door to a broader an deeper study of coarse-grained models aimed at human-intensive tasks. Traditionally, many methodologies have focused on inferring safe, fine-grained models that can be used as input to other techniques such as verification. While these models have proven their value, the safety requirement sometimes comes at the price of restrictions on the input artefact or skyrocketing construction times. This work presents evidence that *i)* coarse-grained models are affordable, and *ii)* they can play a key role in human-intensive development activities.

Finally, we explored coarse-grained behaviour abstractions that are permissive but not safe. We believe that this work can be complemented in two ways:

1. By considering coarse-grained abstractions for artefacts other than APIs. For instance, creating models that developers can use to better understand their programs in terms of their structure or features.

2. By studying coarse-grained behaviour abstractions that are safe but not permissive. Having both kinds of models could help the developer figure out lower and upper bounds for the admissible legal usage of an API.

Bibliography

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical computer science*, 138(1):3–34, 1995.

[AČMN05] R. Alur, P. Černỳ, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL '05*, pages 98–109, 2005.

[BA08] K. Bierhoff and J. Aldrich. Plural: checking protocol compliance under aliasing. In *ICSE*, pages 971–972. ACM, 2008.

[BB04] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16 th International Conference on Computer Aided Verification (CAV '04)*, pages 515–518, 2004.

[BBSE11] I. Beschastnikh, Y. Brun, S.S.M. Sloan, and M.D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *FSE 2011*, 2011.

[BHJM07] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9:505–525, 2007.

[BKA11] Nels E. Beckman, Duri Kim, , and Jonathan Aldrich. An empirical study of object protocols in the wild. In *ECOOP 2011*, 2011.

[BN11] N.E. Beckman and A.V. Nori. Probabilistic, modular and scalable inference of typestate specifications. *PLDI*, 2011.

[CGN+05] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Model-based testing of object-oriented reactive systems with Spec Explorer. *Microsoft Research MSR-TR-2005-59, May*, 2005.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CK05] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. *Lecture Notes in Computer Science*, pages 108–128, 2005.

[CZvD+09] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *TSE*, 35(5):684–702, 2009.

[dCBGU09] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Validation of contracts using enabledness preserving finite state abstractions. In *ICSE '09*, pages 452–462, 2009.

[dCBGU11] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. Program abstractions for behaviour validation. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 381–390, 2011.

[dCBGU12a] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE Transactions on Software Engineering*, 38(1):141 –162, Jan.-Feb. 2012.

[dCBGU12b] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. Abstractions for validation in action. *LNCS*, 7320, 2012.

[DDA12] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *PLDI 2012*, pages 181–192. ACM, 2012.

[DdM06] B. Dutertre and L. de Moura. The Yices SMT solver. *Available a t http://yices. csl. sri. com/, August*, 2006.

[DF01] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *PLDI '01*, pages 59–69, 2001.

[DF04] R. DeLine and M. Fahndrich. Typestates for Objects. *Ecoop 2004-Object-Oriented Programming: 18th European Conference, Oslo, Norway, June, 2004: Proceedings*, 2004.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DKM+10] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *ISSTA 2010*, 2010.

[DLWZ06] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Workshop on Dynamic systems analysis '06*, 2006.

[DMB08] L. De Moura and N. Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[DR09] Brian Demsky and Martin Rinard. Automatic extraction of heap reference properties in object-oriented programs. *IEEE Transactions on Software Engineering*, 35:305–324, 2009.

[EPG+07] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 2007.

[Esp97a] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, February 1997.

[Esp97b] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

[GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *ISSTA '02*, pages 112–122, 2002.

[GJHV95] E. Gamma, R. Johnson, R. Helm, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GKM+08a] W. Grieskamp, N. Kicillof, D. MacDonald, A. Nandan, K. Stobie, and F. Wurden. Model-based quality assurance of Windows protocol documentation. In *ICST '08*, pages 502–506, 2008.

[GKM+08b] Wolfgang Grieskamp, Nicolas Kicillof, Dave MacDonald, Alok Nandan, Keith Stobie, and Fred L. Wurden. Model-based quality assurance of Windows protocol documentation. In *ICST*, pages 502–506. IEEE Computer Society, 2008.

[GKSB11] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.

[GMM09] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *ICSE '09*, pages 430–440, 2009.

[GP09] D. Giannakopoulou and C.S. Păsăreanu. Interface generation and compositional verification in JavaPathfinder. In *FASE '09*, pages 94–108, 2009.

[GPL07] Gnu general public license, version 3. http://www.gnu.org/licenses/gpl.html, June 2007. Last retrieved 2012-05-10.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV '97*, pages 72–83, 1997.

[GS08] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE '08*, pages 51–60, 2008.

[Har87] David Harel. StateCharts: A Visual Formalism for Complex Systems. *Science of Comp. Program.*, 8:231–274, 1987.

[HJL96] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231–261, 1996.

[HJM05] T.A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/FSE '05*, pages 31–40, 2005.

[HMU07] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-wesley, 2007.

[Hod97] Wilfrid Hodges. *A shorter model theory.* Cambridge University Press, Cambridge New York, 1997.

[IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology, September 1990.

[JH08] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *TAIC PART 2008*, pages 94–98. IEEE, 2008.

[JH11] Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5):649 –678, sept.-oct. 2011.

[KFR+95] J. Klensin, N. Freed, M. Rose, E. Stefferud, and D. Crocker. Smtp service extensions. Technical report, RFC 2846, November, 1995.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[KPV03] S. Khurshid, C. PĂsĂreanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, 2003.

[Kra07] Jeff Kramer. Is abstraction the key to computing? *Commun. ACM*, 50:36–42, April 2007.

[KV07] P.H. Kvam and B. Vidakovic. *Nonparametric statistics with applications to science and engineering*, volume 653. John Wiley & Sons, 2007.

[LB08] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(2):185–203, 2008.

[Lin93] J. Linn. RFC1508: Generic Security Service Application Program Interface. *RFC Editor United States*, 1993.

[LKMU08] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engineering Journal*, 15(2):175–206, 2008.

[LMP08] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08*, pages 501–510, 2008.

[LMS07] L. Liu, B. Meyer, and B. Schoeller. Using contracts and boolean queries to improve the quality of automatic test generation. In *TAP '07*, pages 114–130, 2007.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *STOC '92*, pages 264–274, 1992.

[MAV05] C. Metayer, J.R. Abrial, and L. Voisin. Event-b language. *RODIN Project Deliverable D*, 7, 2005.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[MS-08] [MS-NNS]: .NET NegotiateStream Protocol Specification v2.0, July 2008. http://msdn.microsoft.com/en-us/library/cc236723.aspx.

[MS-09a] [MS-PCCRR]: Peer Content Caching and Retrieval: Retrieval Protocol Specification v2.0.1, December 2009. http://msdn.microsoft.com/en-us/library/dd304175(PROT.13).aspx.

[MS-09b] [MS-WINSRA]: Windows Internet Naming Service (WINS) Replication and Autodiscovery Protocol Specification, May 2009. http://msdn.microsoft.com/en-us/library/dd357279%28PROT.10%29.aspx.

[NFLTJ06] C. Nebut, F. Fleurey, Y. Le Traon, and J.M. Jézéquel. Automatic Test Generation: A Use Case Driven Approach. *IEEE TSE*, pages 140–155, 2006.

[NGC05] M.G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. *ACM SIGPLAN Notices*, 40(10):77–96, 2005.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[PCM08] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts, 2008.

[PE07] C. Pacheco and M.D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.

[PG09] Michael Pradel and Thomas R. Gross. Automatic Generation of Object Usage Specifications from Large Method Traces. In *ASE 2009*, pages 371–382. IEEE, November 2009.

[PGKG08] M. Pinzger, K. Gräfenhain, P. Knab, and H.C. Gall. Incremental visual understanding of Java source code. Technical report, U. of Zurich, 2008.

[RT06] S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. *Department of Computer Science, The University of Iowa, Tech. Rep*, 2006.

[SA06] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006.

[SD06] J. Sun and J.S. Dong. Design Synthesis from Interaction and State-Based Specifications. *IEEE TSE*, 2006.

[SDK+11] Raimondas Sasnauskas, Oscar Soria Dustmann, Benjamin Lucien Kaminski, Klaus Wehrle, Carsten Weise, and Stefan Kowalewski. Scalable symbolic execution of distributed systems. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS '11, pages 333–342, Washington, DC, USA, 2011. IEEE Computer Society.

[SHKR12] Matt Staats, Shin Hong, Moonzoo Kim, and Gregg Rothermel. Understanding user understanding: determining correctness of generated program invariants. In *ISSTA 2012*, pages 188–198, New York, NY, USA, 2012. ACM.

[Spi92] J.M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., 1992.

[SY86] RE Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 12(1):157–171, 1986.

[Tca10] Alexis Tcach. Contract simulation via enabledness-based behaviour abstractions. Master's thesis, Departamento de Computación, FCEyN, Universidad de Buenos Aires, 2010.

[TTDBS07] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä. Empirical studies in reverse engineering: state of the art and future trends. *Empirical Software Engineering*, 12(5):551–571, 2007.

[TY01] S. Tripakis and S. Yovine. Analysis of Timed Systems Using Time-Abstracting Bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.

[Uri99] Tomas Uribe. *Abstraction-based Deductive-algorithmic Verification of Reactive Systems*. Stanford University, Dept. of Computer Science, 1999.

[Val98] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer Berlin / Heidelberg, 1998.

[VvLMP04] H.T. Van, A. van Lamsweerde, P. Massonet, and C. Ponsard. Goal-oriented requirements animation. In *Requirements Engineering Conference, 2004.*, pages 218–228, 2004.

[WSCF00] J. Whittle, J. Schumann, N.A.R. Center, and M. Field. Generating statechart designs from scenarios. In *ICSE '00*, pages 314–323, 2000.

[ZBdC+11] Edgardo Zoppi, Víctor Braberman, Guido de Caso, Diego Garbervetsky, and Sebastián Uchitel. Contractor.net: inferring typestate properties to enrich code contracts. In *Proceeding of the 1st workshop on Developing tools as plug-ins*, TOPI '11, pages 44–47, New York, NY, USA, 2011. ACM.

List of Figures

List of Tables