Automated Abstractions for Contract Validation

Guido de Caso, Víctor Braberman, Diego Garbervetsky and Sebastián Uchitel

Abstract—Pre/post condition-based specifications are common-place in a variety of software engineering activities that range from requirements through to design and implementation. The fragmented nature of these specifications can hinder validation as it is difficult to understand if the specifications for the various operations fit together well. In this paper, we propose a novel technique for automatically constructing abstractions in the form of behaviour models from pre/post condition-based specifications. Abstraction techniques have been used successfully for addressing the complexity of formal artefacts in software engineering; however, the focus has been, up to now, on *abstractions for verification*. Our aim is *abstraction for validation* and hence, different and novel trade-offs between precision and tractability are required. More specifically, in this paper, we define and study enabledness preserving abstractions, that is, models in which concrete states are grouped according to the set of operations that they enable. The abstraction results in a finite model that is intuitive to validate and which facilitates tracing back to the specification for debugging. The paper also reports on the application of the approach to two industrial strength protocol specifications in which concerns were identified.

Index Terms—Requirements/Specifications, Validation, Automated Abstraction.

1 Introduction

Pre/post condition specifications constitute good practice in a variety of software engineering activities [1]. In requirements engineering, they provide the link between declarative high-level system goals, and operational requirements for the software-to-be [2]. Use case specifications, which are popular in development processes such as RUP (Rational Unified Process) are also equipped with pre and postconditions. In design, the notion of design by contract [3], as a mechanism to abstract the way functionality is provided by a procedure or method, is underpinned by pre/post conditions. Object oriented design commonly includes design of method pre and postconditions in addition to the specification of class or object invariants. At the code level, the use of assertions to verify at run-time pre/post conditions is considered good-practice [4].

A pre/post condition pair constitutes a specification that is local to a specific operation (method, procedure, use case, event, etc.). The precondition is an assertion that is expected to hold before the occurrence of the operation, the postcondition is an assertion that is guaranteed to hold after the occurrence of the operation if the precondition held before the occurrence. Typically, a contract specification will include various operations (needed to provide some significant service) each with a pre/post condition pair and possibly an invariant that is expected to hold after the occurrence of any sequence of the specified operations

- *All the authors are with the Department of Computing, FCEyN, UBA, Buenos Aires, Argentina.*
  E-mail: {gdecaso, vbraber, diegog, suchitel}@dc.uba.ar
- *Sebastián Uchitel is also with the Department of Computing, Imperial College, London, UK.*
  E-mail: su2@doc.ic.ac.uk

Validating pre/post condition specifications, i.e., understanding if the there is a correspondence between the meaning of the specification and the meaning that the specification was expected to have, is a difficult and error prone task. Although understanding the pre/post condition for a single operation may be relatively simple, understanding if the chaining of pre/post conditions for an arbitrary sequence of operations is describing the intended outcome is complicated. For example, ensuring that the pre/post conditions of a set of operations of an API are correct requires understanding if the set of pre/post conditions of the API preserve the system invariant and fit together adequately to provide the intended API functionality. Validating a use case model requires understanding how the various use-cases can be combined to provide (and only provide) the expected software-wide requirements.

In this paper we propose a strategy for validation of pre/post condition specifications based on the conjecture that pre/post condition specifications would benefit from easily auditable abstractions that exhibit global implications of locally specified behaviour.

Validation techniques can be classified into two complementary strategies. The first is to convert the problem into a verification problem by producing a specification against which the artefact to be validated can be rigorously checked, possibly even (semi-)automatically using testing, model checking, theorem proving or combinations thereof. There are multiple advantages to this strategy and many approaches have shown it to be effective. However, there are downsides too. An alternative specification must be built or be available, and we must be sure that it has been validated appropriately; in other words, the validation problem has been simply shifted.

The second strategy is to automatically produce alternative views of the artefact to be validated so that the engineer can contrast his or her understanding of the

expected behaviour against a different, but semantically equivalent, perspective of the original specification. In other words to transform the problem into a different, hopefully simpler, validation problem. In this strategy, the human plays a crucial role as it is she or he that must identify a mismatch between the intended behaviour and the specified behaviour.

A number of different approaches that fall into this second validation strategy can be used to for pre/post condition specifications. *Manual inspection and review techniques* can identify some problems but are limited in that reviewers must build a cohesive mental model of the overall behaviour of the specification from a fragmented specification in order to understand if: *a)* all admissible sequences of operations (sequences where an operation is applied only if its precondition is met) yield the expected result and *b)* if all sequences of operations that are considered valid are in effect admissible according to the pre/post conditions specified. *State space exploration* is an alternative validation strategy. A state machine can be constructed, possibly automatically, and then be explored to gain confidence on the validity of the pre/post condition specification. One of the main limitations here is the state space of such a model is potentially infinite and certainly unmanageable as a whole for an engineer. Inspection of parts of the state space, via *simulation or animation* is an alternative; but it provides, in addition to a very partial exploration, a localised linear view that may not suffice to detect problems.

In this paper we propose a complementary approach for validation of pre/post condition specifications that is based on the static construction of a conservative abstraction *of the specification in the form of a behaviour model which exhibits the global implications of locally specified behaviour.* We conjecture that validation of abstract behaviour models automatically constructed from pre/post condition specifications can facilitate the validation of the latter.

Behaviour models such as finite state machines and action machines [5] are well founded formalisms that allow describing the temporal relation between the occurrence of events. Depending on the context of use, these events can be interpreted in various ways such as operations, methods, procedures. Behaviour models are used in requirements engineering for providing the expected behaviour of the software-to-be or of external agents that interact with it. These models are also used to explain the expected usage of an API, the expected communication protocol between processes, or to provide an abstract view of the state space of a system and how various operations affect it.

Behaviour models are a popular target for synthesising fragmented behaviour information. They can be synthesised from requirements specifications [6], use cases, and scenarios [7]. Their intuitive graphical representation and their executable semantics makes them good choices for validation. The aim of this work is to support validation of software engineering artefacts that rely on pre/post conditions as a means for specification by automated construction and tool-supported analysis of behaviour models that abstract the state space of these artefacts sufficiently to make validation tractable.

The use of abstract behaviour models for addressing the complexity of formal artefacts in software engineering is not novel. In particular, there has been a significant amount of work in the use of abstractions for *verification* (e.g., [8]) where the aim is to reduce the complexity of the artefact to be verified automatically against some property by automated abstraction. The absence of a violation to the property in the (significantly more tractable) abstraction guarantees that the original artefact satisfies the property. The price to be paid for the abstraction is that of precision: a violation of the property in the abstraction may however be spurious, not corresponding to a behaviour in the original artefact.

The work presented in this paper, however, aims at using *abstraction to support validation* instead of verification. Hence, the level of abstraction required to obtain an appropriate trade-off between precision and tractability is different and the computation of the abstraction poses different challenges. In this paper we present and study a particular level of abstraction, enabledness preserving abstractions, and show that it supports validation of complex, industrial strength, specifications.

More specifically, in this paper, we propose a novel technique for constructing behaviour models from contract specifications, i.e., operations specified with pre- and post conditions. Given a contract, the resulting behaviour model is an abstraction of all possible implementations that satisfy the contract. The level of abstraction chosen to construct the behaviour model can be seen as a generalisation of the pre/post condition philosophy: A precondition describes the state in which a specific operation is permissible, we are interested in capturing the precondition for each arbitrary set of operations. In other words, each state in the resulting behaviour model should characterise the condition for which a subset of the specified operations is enabled; this means that the invariant of the state is the conjunction of the preconditions enabled at that state. The contract abstraction is then completed by adding transitions according to the preconditions and postconditions of the operations they model: A transition can be added if the precondition for the operation holds on the source state and the postcondition holds on the target state.

The models constructed by the approach described herein can be used to validate contract pre/post-condition-based specifications through inspection, animation and simulation. We believe, and our experience so far confirms, that the criteria chosen for abstraction facilitates validation and debugging. Firstly, because a formal and intuitive correspondence exists between the state space of the behaviour model and that of the artefact being specified. Furthermore, that correspondence is structured in a way that can be easily traced back to the original specification. Not only does each state in the behaviour model represent an invariant expressed

CircularBuffer
variable a array of integers
variable wp, rp integer
inv $0 \leq rp < |a| \ \wedge \ 0 \leq wp < |a| \ \wedge \ |a| > 3$
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 $rv = a[rp'] \ \wedge \ wp' = wp \ \wedge \ a' = a$
$\wedge \ (rp < |a| - 1 \ \Rightarrow \ rp' = rp + 1)$
$\wedge \ (rp = |a| - 1 \ \Rightarrow \ rp' = 0)$

Fig. 1. Specification of a circular buffer

in terms of the variables, predicates and propositions that appear in the specification (and hence constructing concrete scenarios from abstract ones is straightforward), but also the invariants are expressed as a conjunction of preconditions, each of which is a building block of the specification being validated (and hence facilitating the identification of problematic operations). Secondly, because in the case studies conducted so far, the state-based models we have produced automatically from contract specifications have had a similar level of abstraction to models manually produced by the authors of the contract specification. For instance, we have produced abstractions that correspond to manually produced typestate specifications for object oriented classes [9], and abstractions that are comparable to the state-machines included in Microsoft technical documents to aid the comprehension of their protocol specifications.

In summary, the contributions of this paper are: i) a definition of finite state abstraction of a pre/post-condition-based specification, ii) the notion of enabledness preserving abstraction as an adequate level of abstraction to support contract validation, iii) an algorithm, and tool, that constructs enabledness preserving abstractions from contracts, which in practice is quadratic with respect to the amount of states of the abstraction and that scales to industrial strength contract specifications, iv) the validation of two industrial strength protocol specifications in which our approach supported the identification of a number of ambiguities and inconsistencies, and v) a number of heuristics based on the structural characteristics of an enabledness preserving behaviour abstractions that can help identify problems in pre/post-condition-based specifications.

The rest of this paper is organised as follows. We begin with a motivation example that informally introduces the concept of contract validation via finite state abstractions (Section 2). We continue with the formal definition of contracts, contract implementations and finite state contract abstractions (Section 3). We then introduce the notion of enabledness equivalence and enabledness-preserving contract abstractions (Section 4). Subsequently, we report on the implementation of our approach and the validation of the tool and approach on four case studies (Section 5). Then we comment on a number of guidelines which proved to be useful when validating contract by means of enabledness-preserving abstractions (Section 6). Finally, we discuss related work (Section 7), ideas for future work (Section 8) and conclusions (Section 9).

2 MOTIVATION

In this section, we motivate our approach by illustrating the difficulties of validating pre/post condition specifications using a toy example.

Consider the specification of a circular buffer given in Figure 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, case in which 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 for a circular buffer is? 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 [10] and in particular infinite state model checking [11] allow verifying if the the entire state-machine defined by a contract specification, as described above, satisfies a property. Theorem proving allows checking if a property can be directly inferred

1. Notice however, that the actual storing capacity is always reduced by one.

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, of the circular buffer in this case, twice: Once in an operational pre/post condition style and the other in a, for instance, more declarative style.

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

Instead, the complementary approach we propose is to automatically construct a behaviour model such as the one shown in Figure 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 abstract state `S0`, concrete states that allow `write` and `read` are grouped into abstract state `S1`, and all concrete states that only allow to `read` are abstracted into `S2`. Transitions between abstract states exist only if a transition between concrete states they represent exist. Finally, an abstract state is an initial state (marked with a double circle) if it abstracts at least one initial concrete state of the circular buffer.

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 providing further support for analysis and validation of pre/post specifications. The model of the circular buffer specification shown in Figure 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, full, or partially full/empty: State 0 represents a buffer in which we can write but we cannot read, state 1 allows both actions to be performed and state 2 allows reading only.

Consider the `write`-labelled transition from state 1 to 0. 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 1 (non-full) to state 2 (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.

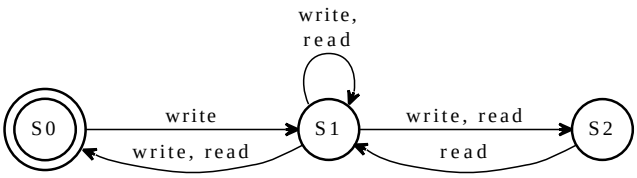

Fig. 2. Circular buffer finite abstraction

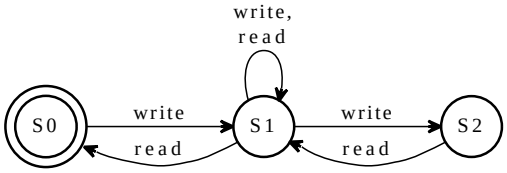

Fig. 3. Corrected circular buffer finite abstraction

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 and the specification. The concrete states of the circular buffer are formally abstracted to get the model in that figure according to following invariants:

- State 0: `inv` $\wedge$ `write_pre` $\wedge$ $\neg$`read_pre`
- State 1: `inv` $\wedge$ `write_pre` $\wedge$ `read_pre`
- State 2: `inv` $\wedge$ $\neg$`write_pre` $\wedge$ `read_pre`

Let us now try to understand why the transition labelled `write` from states 1 to 0 appears in Figure 2 and if this is signalling a problem in the specification. The fact that the transition is enabled in state 1 follows directly from the choice of level of abstraction of Figure 2. State 1 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 0. The question can be answered by asking how can the invariant of state 0 hold if the invariant of state 1 holds and action `write` occurs; question which can be easily answered automatically with appropriate tool support: If $rp = wp$ holds on top of the invariant for state 1, then the postcondition for `write` leads to state 0.

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 3) without the two suspicious transitions described previously. 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 actions starting from the initial state guarantees $rp \neq wp$ yet the omission becomes a problem if the buffer is extended

ExtendedCircularBuffer

$\vdots$

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

$\vdots$

action `reset()`
pre true
post $rp' = wp \ \wedge \ wp' = wp \ \wedge \ a' = a$

Fig. 4. Circular buffer with `reset`

with legal operations (those that preserve the incomplete invariant) such as the specification shown in Figure 4.

In summary, the example above illustrates how the depiction of an abstract model that integrates the various pieces of information that appear in a contract specification supports validation of such specifications 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. In the rest of the paper we present a novel technique to automatically construct abstract behaviour models like the one in Figure 2 from contracts such as the ones depicted in Figures 1 and 4 and discuss validation of our approach on a number of case studies.

3 Finite State Contract Abstractions

In this section, we define the formal underpinnings of the problem we want to solve: finding a finite abstraction of a contract. Firstly, we define contracts and their meaning as a set of acceptable implementations. Then, we define abstractions as a finite state machine which is able to simulate any valid implementation of the protocol.

We call $\mathbb{P}(X)$ the set of first order predicates whose free variables are included in X. We will use the operator X' to refer to the set of variables $\{ x' \mid x \in X \}$.

Definition 1 (Contract)**.** *A structure of the form* $C = \langle V,\ inv,\ init,\ A,\ P,\ Q \rangle$*, is called a* contract *when:*

- V *is a finite set of* variables*.*
- $inv \in \mathbb{P}(V)$ *is the* system invariant*.*
- $init \in \mathbb{P}(V)$ *is the* initial predicate*.*
- $A = \{ a_1, \ldots, a_n \}$ *is a finite set of* action labels*.*
- $P : A \to \mathbb{P}(V \cup \{ p \})$ *is a total mapping that assigns a precondition for each of the action labels. Note that the distinguished variable* p *stands for the name of any action parameter*[2]*.*
- $Q : A \to \mathbb{P}(V \cup V' \cup \{ p \})$ *is a total mapping that assigns a postcondition for each of the action labels, where* v' *stands for the new value of the variable* v *after an action execution.*

Example 1. *Formally, the specification given in Figure 1 denotes the contract* $C = \langle V,\ inv,\ init,\ A,\ P,\ Q \rangle$ *where:*

$$
\begin{aligned}
V &= \{ a, rp, wp \} \\
inv &= 0 \leq rp < |a| \ \wedge \ 0 \leq wp < |a| \ \wedge \ |a| > 3 \\
init &= |a| > 3 \ \wedge \ rp = |a| - 1 \ \wedge \ wp = 0 \\
A &= \{ \texttt{read}, \texttt{write} \}
\end{aligned}
$$

$$
\begin{aligned}
P_{\texttt{write}} &= (wp < rp - 1) \ \vee \ (wp = |a| - 1 \ \wedge \ rp > 0) \\
&\quad \vee \ (wp < |a| - 1 \ \wedge \ rp < wp) \\
Q_{\texttt{write}} &= rp' = rp \ \wedge \ (wp = |a| - 1 \ \Rightarrow \ wp' = 0) \\
&\quad \wedge \ (wp < |a| - 1 \ \Rightarrow \ wp' = wp + 1) \\
&\quad \wedge \ (a' = \mathrm{updateArray}(a, wp, n)) \\
P_{\texttt{read}} &= (rp < wp - 1) \ \vee \ (rp = |a| - 1 \ \wedge \ wp > 0) \\
&\quad \vee \ (rp < |a| - 1 \ \wedge \ wp < rp) \\
Q_{\texttt{read}} &= \exists \ rv \ \big(rv = a[rp'] \ \wedge \ wp' = wp \ \wedge \ a' = a \\
&\quad \wedge \ (rp < |a| - 1 \ \Rightarrow \ rp' = rp + 1) \\
&\quad \wedge \ (rp = |a| - 1 \ \Rightarrow \ rp' = 0) \big)
\end{aligned}
$$

Notice that the translation is straightforward except for the return values, which are existentially quantified in the postcondition. We do not take into consideration the return values because we are only interested in the effects that the actions have on the system variables.

On the other hand, contract implementations will be defined on top of what we call *Data State Machine* (which is a sort of simplified version of an Action Machine [5]). Data State Machines have states labelled by mappings from variable names to a given value domain while transitions are labelled with actions together with actual parameter values.

Definition 2 (Data State Machine (DSM))**.** *A structure of the form* $I = \langle \mathcal{V},\ \mathcal{D},\ \mathcal{A},\ \mathcal{S},\ \mathcal{S}_0,\ \Delta \rangle$*, is called* Data State Machine *when:*

- $\mathcal{V}$ *is a finite set of variable names.*
- $\mathcal{D}$ *is a value domain.*
- $\mathcal{A}$ *is a set of* action labels*.*
- $\mathcal{S}$ *is a set of states denoted by functions from* $\mathcal{V}$ *to* $\mathcal{D}$ *(i.e.,* $\mathcal{S} \subseteq \mathcal{V} \to \mathcal{D}$*).*
- $\mathcal{S}_0 \subseteq \mathcal{S}$ *is the set of initial states.*
- $\Delta : \mathcal{S} \times \mathcal{A} \times \mathcal{D} \to \wp(\mathcal{S})$ *is a transition function.*

Now we define an implementation of a contract as a DSM that satisfies the contract:

Definition 3 (Contract Implementation)**.** *Given a contract* $C = \langle V,\ inv,\ init,\ A,\ P,\ Q \rangle$*, a value domain* $\mathbb{D}$ *and an interpretation* $\mathbb{D}^{op}$ *for the symbols appearing in predicates. We say that a Data State Machine of the form* $I = \langle \mathcal{V},\ \mathcal{D},\ \mathcal{A},\ \mathcal{S},\ \mathcal{S}_0,\ \Delta \rangle$ *is an* implementation for the contract C *under the interpretation* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ *iff the following hold:*

1) $\mathcal{V} \supseteq V$, $\mathcal{D} = \mathbb{D}$, $\mathcal{A} = A$.
2) $init(s)$ *yields true for each* $s \in \mathcal{S}_0$.

2. For the sake of simplicity, and without loosing generality, we set the number of parameters to 1. More parameters could be accommodated by thinking of p as the name of a n-uple.

3) *There exists a set of states $\mathcal{S}_v \subseteq \mathcal{S}$ such that $inv(s)$ yields true for each $s \in \mathcal{S}_v$, $\mathcal{S}_0 \subseteq \mathcal{S}_v$ and for each $a_i \in A$ and $d \in \mathcal{D}$ such that $P_{a_i}(s \cup \{ p \mapsto d \})$ yields true then $\Delta(s, a_i, d)$ is non-empty and each state $s' \in \Delta(s, a_i, d)$ is also included in $\mathcal{S}_v$. Furthermore, $Q_{a_i}(s \cup s' \cup \{ p \mapsto d \})$ holds*[3].

Conceptually, an implementation is a legal operationalization of the actions described by a contract. This operationalization must satisfy that the initial configurations are allowed by the contract and that every time that it evolves then it follows the pre and postconditions established by the contract.

In the rest of the paper, given an implementation, $\mathcal{S}_v$ will denote the smallest set satisfying the above conditions.

Example 2. *A possible implementation of contract of Figure 1 for a buffer of size four is the Data State Machine of the form $I = \langle \mathcal{V},\ \mathcal{D},\ \mathcal{A},\ \mathcal{S},\ \mathcal{S}_0,\ \Delta \rangle$, where:*

$$
\begin{array}{rcl}
\mathcal{V} & = & \{ a, wp, rp \} \\
\mathcal{D} & = & \mathbb{Z} \cup (\{ 0,1,2,3 \} \to \mathbb{Z}) \\
\mathcal{A} & = & \{ \texttt{read},\ \texttt{write} \} \\
\mathcal{S} & = & \left\{ s \;\middle|\; \begin{array}{c} |s(a)| = 4 \ \wedge\ 0 \le s(rp) < 4 \ \wedge \\ 0 \le s(wp) < 4 \ \wedge s(rp) \neq s(wp) \end{array} \right\} \\
\mathcal{S}_0 & = & \left\{ \left(\begin{array}{c} a \mapsto [0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 0], \\ rp \mapsto 3,\ wp \mapsto 0 \end{array} \right) \right\}
\end{array}
$$

Informally, the function Δ is defined as having transitions from every state that satisfies the precondition of a given action, going to every possible state that satisfies the post-condition of the same action.

The number of states of this implementation is $values^{|a|} \times |a| \times (|a| - 1)$, where $values$ is the number of different values that can be entered in the array. For instance, if we only allow boolean elements and the array is of size 4, we would have 192 states. This shows that abstraction is necessary even for a simple example like the circular buffer.

We use Finite State Machines to provide an abstract representation of a contract, or more precisely, of the implementations allowable by a contract. Simply, a *FSM* is defined as 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 $\delta : S \times \Sigma \to \wp(S)$ is a transition function.

We now define a finite contract abstraction as a FSM which is able to simulate any possible contract implementation.

Definition 4 (Finite State Contract Abstraction (FSCA))**.** *Given a contract $C = \langle V,\ inv,\ init,\ A,\ P,\ Q \rangle$, an interpretation $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ and a FSM $M = \langle S,\ S_0,\ \Sigma,\ \delta \rangle$, we say that M is a* finite state contract abstraction *(FSCA) of C under the interpretation $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ iff for each implementation $I = \langle \mathcal{V},\ \mathcal{D},\ \mathcal{A},\ \mathcal{S},\ \mathcal{S}_0,\ \Delta \rangle$ of C there exists a total function $abs_I : \mathcal{S}_v \to S$, called* witness abstraction function*, such that:*

1) $abs_I(\mathcal{S}_0) \subseteq S_0$
2) *For every $s \in \mathcal{S}_v$, and every action label a_i and actual parameter d such that P_{a_i} holds, then $abs_I(\Delta(s, a_i, d)) \subseteq \delta(abs_I(s), a_i)$.*

Having fixed, in the notion of contract, what we mean by a pre/post-condition-based specification, and having formally defined contracts, their acceptable implementations and finite state abstractions of these, in the next section we concentrate on finding a finite state abstraction of a contract which is abstract enough to enable validation yet not too coarse (note that universal language generator would fit previous definition) to impede finding problems with the contract-under-analysis.

Moreover, an FSCA is able to reproduce any legal action sequence allowed by a contract.

Lemma 1. *Given a contract $C = \langle V,\ inv,\ init,\ A,\ P,\ Q \rangle$, an interpretation $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ and its FSCA $M = \langle S,\ S_0,\ \Sigma,\ \delta \rangle$, then every path on any implementation I of C is accepted by the language of M.*

4 FSCAs FOR CONTRACT VALIDATION

In this section, we show how to construct a finite state contract abstraction from a contract. The particular level of abstraction for the FSMs to be constructed is based on the notion of *enabledness*. This level of abstraction results in state invariants in the contract abstraction which are compact, intuitive and can be easily traced back to the contract. We believe that this is essential to facilitate the task of the engineer that must mentally fill the gap between abstraction and contract in order to validate and fix the latter.

The core idea for setting the level of abstraction to support contract validation is to capture the different states of the contract that are relevant in terms of the operations which are enabled at a given time. This means that we will group together concrete states of contract implementations based on the preconditions that are satisfied at those states.

Definition 5 (Enabledness Equivalence)**.** *Given a contract $C = \langle V,\ inv,\ init,\ A,\ P,\ Q \rangle$, an implementation of the form $I = \langle \mathcal{V},\ \mathcal{D},\ \mathcal{A},\ \mathcal{S},\ \mathcal{S}_0,\ \Delta \rangle$ of C under $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ and two concrete states $s, t \in \mathcal{S}$ we say that s and t are* enabledness equivalent *states (noted $s \equiv_e t$) iff for every $a \in \mathcal{A}$:*

- $\exists\ d\ .\ P_a(s \cup \{ p \mapsto d \}) \ \Rightarrow\ \exists\ d'\ .\ P_a(t \cup \{ p \mapsto d' \})$
- $\exists\ d'\ .\ P_a(t \cup \{ p \mapsto d' \}) \ \Rightarrow\ \exists\ d\ .\ P_a(s \cup \{ p \mapsto d \})$

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

An *enabledness-preserving abstraction* is a finite state contract abstraction in which concrete states are partitioned by enabledness equivalence. In other words, they are grouped based on the one-step availability of actions.

3. Note that $s' : \mathcal{V} \to \mathcal{D}$, however it can be straightforwardly reinterpreted as a mapping from V' to $\mathbb{D}$.

Definition 6 (Enabledness-preserving FSCA)**.** *A Finite State Contract Abstraction* $M = \langle S,\ S_0,\ \Sigma,\ \delta \rangle$ *of a contract* $C = \langle V,\ inv,\ init,\ A,\ P,\ Q \rangle$ *under an interpretation* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ *is* enabledness-preserving *iff for every implementation* I *of* C *there exists* $abs_I : \mathcal{S}_v \to S$ *(a witness abstraction function) such that given a pair of concrete states* s, t *on* $\mathcal{S}_v$*, then* $s \equiv_e t \ \Leftrightarrow\ abs_I(s) = abs_I(t)$ *holds.*

The previous definition characterises those FSCAs whose state set complies with the enabledness equivalence partition. In order to construct such abstraction, we first need to define a couple of concepts. The first of them is the notion of *action set invariant*. Given a subset of actions as of a contract C, we wish to characterise all concrete states s of implementations of C that satisfy the contract invariant inv in which every action a in as is possible from s (there exists a parameter p for every action a in as such that the precondition P_a of action a holds) and, importantly, in which every action a *not* in as it is *not* possible from s.

Definition 7 (Invariant of an Action Set)**.** *Given a contract* $C = \langle V,\ inv,\ init,\ A,\ P,\ Q \rangle$*, the* invariant of a set of actions $as \subseteq \wp(A)$ *is the predicate* $\mathrm{inv}_{as} \in \mathbb{P}(V)$ *defined as:*

$$\mathrm{inv}_{as} \stackrel{def}{=} inv \ \wedge \bigwedge_{a \in as} \exists p.\ P_a \ \wedge \bigwedge_{a \notin as} \nexists p.\ P_a$$

We can now construct an enabledness-preserving FSCA of a contract by fixing the states to be the enumeration of all the possible action sets. We connect two action sets whenever there is a variable assignment satisfying the invariant of the first set, that executing an action, reaches a variable assignment which satisfies the invariant of the second set.

Theorem 1 (FSCA characterisation)**.** *Given a contract* $C = \langle V,\ inv,\ init,\ A,\ P,\ Q \rangle$ *and an interpretation* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$*, the FSM* $M = \langle S,\ S_0,\ \Sigma,\ \delta \rangle$ *is an enabledness-preserving FSCA of* C *where:*

1) $S = \wp(A)$
2) $as \in S_0$ *iff* $\mathrm{inv}_{as} \Rightarrow$ *init.*
3) $\Sigma = A$.
4) *For all* $as \in S$ *and* $a \in \Sigma$*, if* $a \notin as$ *then* $\delta(as, a) = \emptyset$*, otherwise:*

$$\delta(as, a) \supseteq \{\ bs \ \mid \mathrm{inv}_{as} \ \wedge Q_a \ \wedge \ \mathrm{inv}'_{bs} \textit{ is satisfiable} \ \}$$

The proof for the theorem can be done showing that, given an implementation I,

$$abs_I(s) \stackrel{\text{def}}{=} \{\ a \mid \exists\ d \in \mathbb{D}\ .\ P_a(s \ \cup \ \{\ p \ \mapsto \ d\ \})\ \}$$

is a witness abstraction function satisfying: *i)* Definition 4 and *ii)* that every pair of concrete states s, t satisfy that $s \equiv_e t \ \Leftrightarrow\ abs_I(s) = abs_I(t)$. The first condition can be proved by using the fact that $\mathrm{inv}_{abs_I(s)}(s) = \mathrm{true}$. The second condition is implied by Definition 5.

Returning to the example of Section 2, the FSCA in Figure 2 is an enabledness-preserving abstraction of the circular buffer contract depicted in Figure 1. The action sets for states $S0$, $S1$ and $S2$ are $\{\ write\ \}$, $\{\ write, read\ \}$, and $\{\ read\ \}$, respectively. In addition, it is simple to show that the initial state has been set correctly as init implies $\mathrm{inv}_{\{\ write\ \}}$. The satisfiability proofs for transitions are more complex to show and were computed using SMT solvers (see the next section).

Notice that the abstractions that we produce are able to simulate every possible implementation of a contract. However there may be traces of the FSCA that are not feasible on any given implementation. For instance, `write` → `read` → `read` can be performed in the FSCA of Figure 2 but it is not possible to read twice after writing once on any circular buffer implementation independently of its size.

It is important to note that item 4 of Theorem 1 could be strengthened by requiring equality rather than inclusion. The reason for choosing a weaker condition is that in practice it is undecidable to check if $\mathrm{inv}_{as} \wedge Q_a \wedge \mathrm{inv}'_{bs}$ is satisfiable. The theorem above guarantees that choosing to add transitions in the face of uncertainty still guarantees the construction of a proper abstraction. In the case of the abstraction for the circular buffer in Figure 2 no additional transitions due to unfinished satisfiability checks were added.

In the presence of spurious transitions the possibility of having FSCA traces that are not feasible on any implementation is even higher. Fortunately, state-of-the-art theorem provers are increasingly able to deal with different "kinds" of formulae in a complete fashion and therefore cases of uncertainty did not arise in any of our case studies.

4.1 Construction algorithm

In this section we present an algorithm for the generation of an enabledness-preserving FSCA out of a contract. A trivial algorithm using the concepts of Theorem 1 would require $\Omega(2^n \times n \times 2^n)$ satisfiability queries, where n is the amount of actions in a contract. This is because we would have to consider 2^n states and each state could potentially advance using any of the n actions to any state. Space consumption would also be exponential since the set of states would have to be kept in memory while computing transitions.

Using this *naïve* implementation, together with a inconsistent state pruning phase, was enough for most of the case studies we present here in Section 5. This implementation, together with the execution times, was reported in in [12] without mentioning the inconsistent state pruning phase. Unfortunately the biggest case study, which has 33 actions, was intractable with this implementation and introduced the need for a more efficient algorithm to construct FSCAs.

The idea behind the algorithm we present in this section is splitting the construction problem in 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 FSCA. 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 8 (Enabledness Dependencies)**.** *Given a contract $C = \langle V,\ inv,\ init,\ A,\ P,\ Q\rangle$, we define the following* enabledness dependency relationships *in $A \times A$:*

- $D^{++} \stackrel{def}{=} \{\ (a,b)\ \mid inv \wedge P_a \Rightarrow P_b\ \}$
- $D^{+-} \stackrel{def}{=} \{\ (a,b)\ \mid inv \wedge P_a \Rightarrow \neg P_b\ \}$
- $D^{-+} \stackrel{def}{=} \{\ (a,b)\ \mid inv \wedge \neg P_a \Rightarrow P_b\ \}$
- $D^{--} \stackrel{def}{=} \{\ (a,b)\ \mid inv \wedge \neg P_a \Rightarrow \neg P_b\ \}$

Given a set of actions we will say that it is compliant with the enabledness dependencies relationships if it satisfies all the restrictions that they impose.

Definition 9 (Enabledness Dependencies Compliance)**.** *Given a contract $C = \langle V,\ inv,\ init,\ A,\ P,\ Q\rangle$ and its enabledness dependencies relationships $D^{++}, D^{+-}, D^{-+}, D^{--}$, we say that a set of actions as $\in \wp(A)$* complies with the enabledness dependencies *if all the following conditions hold:*

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

The enabledness dependencies relationships are straightforwardly computed using the following algorithm.

Definition 10 (Enabledness Dependencies Computation Algorithm)**.** *Given a contract $C = \langle V,\ inv,\ init,\ A,\ P,\ Q\rangle$, we construct the enabledness dependencies relationships D^{++}, D^{+-}, D^{-+} and D^{--} using the following procedure.*

1: $D^{++} \leftarrow \emptyset$
2: $D^{+-} \leftarrow \emptyset$
3: $D^{-+} \leftarrow \emptyset$
4: $D^{--} \leftarrow \emptyset$
5: ***for*** $a \in A$ ***do***
6: ***for*** $b \in A$ ***do***
7: ***if*** $inv \wedge P_a \Rightarrow P_b$ ***then***
8: $D^{++} \leftarrow D^{++} \cup \{\ (a,b)\ \}$
9: ***else if*** $inv \wedge P_a \Rightarrow \neg P_b$ ***then***
10: $D^{+-} \leftarrow D^{+-} \cup \{\ (a,b)\ \}$
11: ***end if***
12: ***if*** $inv \wedge \neg P_a \Rightarrow P_b$ ***then***
13: $D^{-+} \leftarrow D^{-+} \cup \{\ (a,b)\ \}$
14: ***else if*** $inv \wedge \neg P_a \Rightarrow \neg P_b$ ***then***
15: $D^{--} \leftarrow D^{--} \cup \{\ (a,b)\ \}$
16: ***end if***
17: ***end for***
18: ***end for***

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.

Lemma 2. *Given a state $s \in \wp(A)$, if inv_s is satisfiable then s, understood as a set of actions, is compliant with the enabledness dependencies relationships.*

Notice that the converse is not true: there exist states that are compliant with the enabledness dependencies relationships but are not consistent. For instance, take a contract with integer variables x, y, z, true as invariant and actions a_1, a_2, a_3 with preconditions $x < y$, $y < z$ and $z < x$ respectively. There are no precondition dependencies, therefore the state $\{\ a_1, a_2, a_3\ \}$ is compliant, but its state invariant is still not satisfiable.

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

Definition 11 (Enumerating States that Comply with Enabledness Dependencies)**.** *Given a contract $C = \langle V,\ inv,\ init,\ \{\ a_1, \ldots, a_n\ \},\ P,\ Q\rangle$ and its enabledness dependencies relationships D^{++}, D^{+-}, D^{-+} and D^{--}, we compute a set of states $S^\star$ given as the result of* ENUM$(\emptyset,\ 1)$.

1: ***procedure*** ENUM(*current*, *i*)
2: ***if*** $i > n$ ***then***
3: ***if*** $\mathrm{inv}_{current}$ *is consistent* ***then***
4: ***return*** $\{\ current\ \}$
5: ***else***
6: ***return*** $\emptyset$
7: ***end if***
8: ***else if*** $a_i \notin current\ \wedge\ (\neg a_i) \notin current$ ***then***
9: $c_1 \leftarrow current \cup \{\ a_i\ \}$
10: $c_1 \leftarrow c_1 \cup \{\ b\ \mid (a_i, b) \in D^{++}\ \}$
11: $c_1 \leftarrow c_1 \cup \{\ (\neg b)\ \mid (a_i, b) \in D^{+-}\ \}$

```
        c2 ← current ∪ { (¬a_i) }
        c2 ← c2 ∪ { b | (a_i, b) ∈ D^{-+} }
        c2 ← c2 ∪ { (¬b) | (a_i, b) ∈ D^{--} }
        return ENUM(c1, i + 1) ∪ ENUM(c2, i + 1)
    else
        return ENUM(current, i + 1)
    end if
end procedure
```

This recursive algorithm uses a set of literals current to hold all those actions that need to be enabled (or disabled) at a certain point. At every step, it analyses the current action a_i and it checks if it is already contained in current, either negated or not.

1) If both a_i and $(\neg a_i)$ are not included in current, the algorithm advances recursively by separately considering the case in which a_i is added to the current set and the case in which $(\neg a_i)$ is added to the current set.
   Before actually performing the recursive call, the algorithm uses the enabledness dependency information to see if any other restrictions can be added to the current set. Due to the nature of the enabledness dependencies, this step could never include any inconsistent restriction (namely the negation of a literal which is already included) into the current set. Formally, for each j such that $j < i$ then:
   - if $(a_i, a_j) \in D^{++}$ then $(\neg a_j) \notin$ current
   - if $(a_i, a_j) \in D^{+-}$ then $a_j \notin$ current
   - if $(a_i, a_j) \in D^{-+}$ then $(\neg a_j) \notin$ current
   - if $(a_i, a_j) \in D^{--}$ then $a_j \notin$ current

   After the recursive calls finished enumerating both set of states, these are joined and returned.
2) If either a_i or $(\neg a_i)$ are contained in current, then it advances to analyse the following action.

If it reaches the end, then it returns the current state (only if its invariant is satisfiable)[4].

Lemma 3. *The set of candidate states $S^\star$, as constructed in Definition 11, satisfies that it is equal to the set of consistent states that comply with the enabledness dependencies relationships.*

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.

Definition 12 (Enabledness-preserving FSCA Construction Algorithm)**.** *Given an input contract $C = \langle V, inv, init, A, P, Q\rangle$ and a set of candidate states S, we build a FSM $M = \langle S, S_0, \Sigma, \delta\rangle$ using the following procedure:*

```
S0 ← { s ∈ S | inv_s ⇒ init }
Σ ← A.
δ(s, a) ← ∅   ∀s, a
for s, s' ∈ S do
    for each action a ∈ s do
        if inv_s ∧ Q_a ∧ inv'_{s'} is satisfiable then
            δ(s, a) ← δ(s, a) ∪ { s' }
        end if
    end for
end for
```

In this algorithm we test each of the candidate states in S to see if they are initial states (which requires $|S|$ logical queries). We then initialize 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 $|S|^2 \times |A|$ logical queries).

We can now postulate that the enabledness-preserving FSCAs constructed by our algorithm are in fact compliant with Definition 6.

Theorem 2. *Given a contract C and an interpretation $\langle \mathbb{D}, \mathbb{D}^{op}\rangle$, then M as built by Definition 12 using the set $S^\star$ of candidate states as given by Definition 11 is an enabledness-preserving FSCA of C under the $\langle \mathbb{D}, \mathbb{D}^{op}\rangle$.*

The proof for this theorem is based on the fact that the algorithm in Definition 12 performs an exhaustive exploration which complies with Theorem 1 over the set of states $S^\star$. All the states which are left out of this exploration (namely, $\wp(A) \setminus S^\star$) would never be part of the final result since their state invariants are insatisfiable, as implied by Lemmas 2 and 3.

The final FSCA construction phase, which is the restriction of the resulting abstraction to its reachable fragment, is entirely standard and will not be analysed in this paper. 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 FSCA construction process. As mentioned before, the enabledness dependencies calculation needs $O(n^2)$ logical 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 can be observed in Table 2 in Section 5. Finally, as mentioned before, the transition calculation phase requires $O(|S^\star|^2 \times n)$ queries.

5 Tool Support and Case Studies

In this section, we comment on some of the aspects involved in the validation of our approach. We discuss tool support and various case studies.

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

5.1 Tool Support

In order to validate our approach, we built a tool called CONTRACTOR[5] that takes a contract description as input and returns an enabledness-preserving finite state contract abstraction. It uses Satisfiability Modulo Theories (SMT) solvers [13] to reason about satisfiability of the formulae as described in Section 4. In cases where these solvers time-out or return "unknown", we assume that the formula is satisfiable, resulting in additional transitions in the enabledness-preserving FSCA. As discussed in the previous section, item 4 of Theorem 1 allows these conservative decisions, guaranteeing the construction of a proper abstraction of the contract. In any case, in all case studies we conducted, no transitions were added as a result of limitations of the SMT solvers (using theories such as linear arithmetic and arrays).

Notice that both the algorithms in Definition 10 and Definition 12 can be easily adapted to make use of multiple worker threads and our CONTRACTOR tool currently implements this.

Furthermore, the algorithm in Definition 10, which calculates the enabledness dependencies relationships, is implemented with a few optimizations in CONTRACTOR. In a first round, only dependencies among labels a_i and a_j with $i \leq j$ 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 satisfiability queries that need to be solved in order to compute the relationships D^{++}, D^{+-}, D^{-+} and D^{--}.

The rest of this section presents four case studies used to test CONTRACTOR, its capabilities and, more importantly, to validate the approach. Note that CONTRACTOR is capable of dealing with complex case studies in times that range from a few seconds up to a couple of minutes in a standard desktop computer (Intel Core i7 with 4GB of RAM memory).

5.2 WebFetcher

The purpose of this case study was to compare the enabledness-preserving abstractions automatically constructed by CONTRACTOR with manually constructed abstractions aimed at static-time reasoning about programs. We considered a case study presented in [9] 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 5.

CONTRACTOR applied to the web fetcher contract results in the FSCA depicted in Figure 6. The states, the transitions (as depicted in the diagram) and the invariants (as computed according to Definition 7) that our technique produces coincide with the manually constructed typestate FSM diagram shown in [9].

5. The CONTRACTOR tool is available on-line together with the contracts used in this section at http://lafhis.dc.uba.ar/contractor.

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

Fig. 5. Specification of a web page fetcher

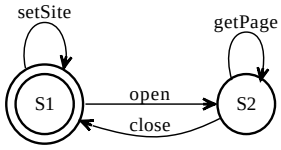

Fig. 6. FSCA for the web page fetcher

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 FSCA construction technique here presented could be used to produce typestates, in the sense of [9], automatically from contracts.

5.3 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 [7] where a statechart [14] 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.

We fed CONTRACTOR with the pre/post specification provided in [7] and obtained the enabledness-preserving FSCA in Figure 7. Note that we did not use the scenarios provided in [7].

We then compared, with respect to simulation [15], the FSCA model with the statechart provided in Figure 11 of [7]. As a result, we found that the following trace in the statechart can not be exhibited by the enabledness-preserving contract abstraction: `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 our FSCA. Based on this observation, we com-

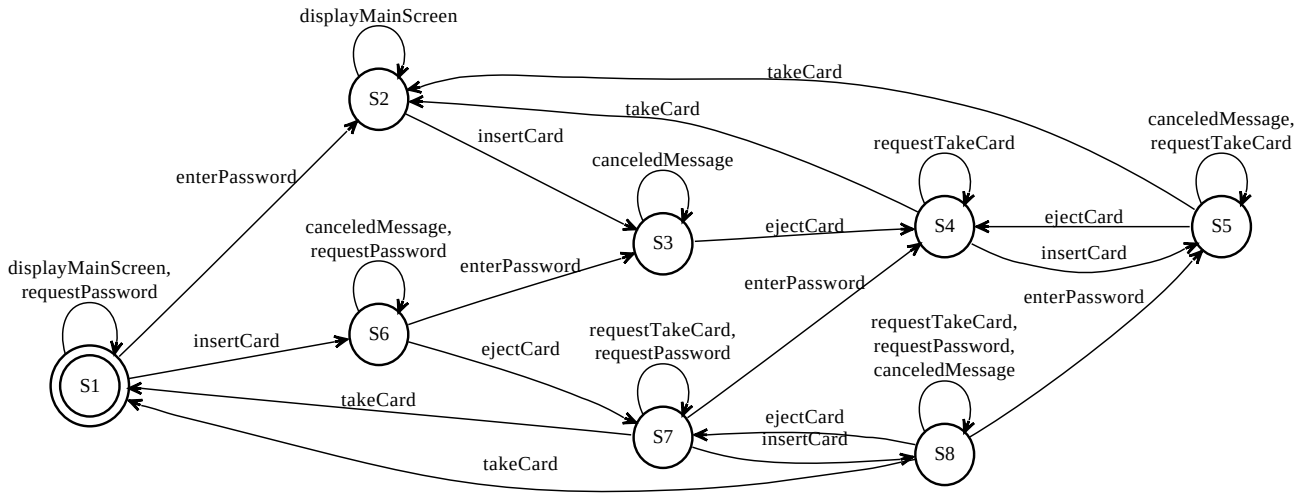

Fig. 7. Enabledness-preserving Finite State Contract Abstraction for the ATM

pared the invariants of the states reached by the execution up to `takeCard` in the FSCA and the statechart. We found that they differed on the acceptable values for the $passwdGiven$ system variable. In the invariant for the FSCA state, $passwdGiven$ is required to be true, while in the invariant for the statechart state, $passwdGiven$ is required to be false. Further analysis shows that `takeCard`'s postcondition does not update the $passwdGiven$ system variable to false. The impact of this omission is that, according to the pre/post specification in [7], 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 [7] does not guarantee preservation of postconditions in the synthesised statechart.

In summary, the construction of an enabledness-preserving FSCA from the pre/post specification of an ATM in [7] supported uncovering errors in the specification and problems with the actual synthesis algorithm therein proposed.

5.4 .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 [16] as a starting point, we selected as case study subject a Microsoft protocol specification currently under revision: The MS-NSS protocol [17] 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 [18] 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 the client the end of the first phase (a security mechanism like Kerberos may have been negotiated run-time). 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 8) was as follows. First, a person completely unfamiliar with the protocol but experienced in writing pre/post condition-based specifications read the publicly available protocol specification document describing the protocol [17]. Then, the same person wrote a contract for the protocol validating the protocol against the document and using as sole automated support the CONTRACTOR tool described above.

Once the protocol's contract was completed, an engineer with experience in protocol validation analysed the enabledness-preserving abstraction produced by CONTRACTOR in order to validate the contract specification and the protocol specification document itself.

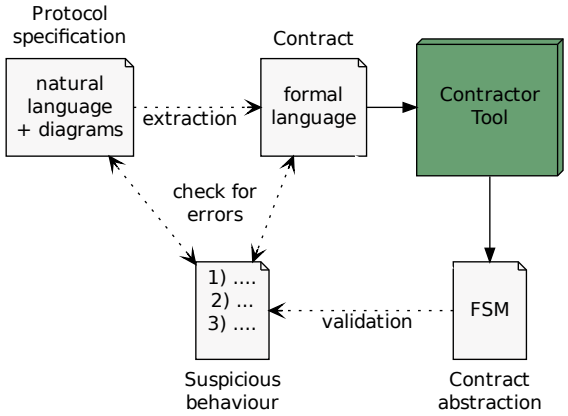

Fig. 8. Experimental setup for the MS-NSS case study

The protocol specification document is structured natural language description containing two auxiliary state machines. The specification states that the natural language description is to be considered the normative specification of the protocol while the state machines are simply aids and references for the reader.

The protocol contract developed was included a set of controllable and observable actions appearing in the specification of client side of the protocol. Only the information provided in natural language was used as a source for the contract-to-be. For instance, Figure 9 depicts a natural language fragment of the original technical document, together with its contract translation. It is worth mentioning that models developed in [17] include server and client-side requirements since the main goal of the QA project is to check protocol specification document compliance against Windows products. For our experiment, the modeller did only resort 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 `SendHandShakeError()`
pre tcpConnection $\wedge$ handShakeState = Proccessed $\wedge$ gssReturned = Error
post (handShakeState' = NotStarted $\wedge$ tcpConnection') $\vee$ (handShakeState' = Error $\wedge$ $\neg$tcpConnection')

Fig. 9. MS-NSS documentation fragment and corresponding translation

During the contract development process the modeller used the enabledness-preserving FSCA produced by CONTRACTOR to eliminate bugs and typos from the specification being developed: The FSCA was analysed using: *i)* inspection techniques, *ii)* simulating scenarios appearing in the protocol specification document, *iii)* checking for bisimilarity of the FSCA against the auxiliary client side state machine of the protocol specification document, and *iv)* composing the FSCA in parallel with the the server side auxiliary state machine of the protocol specification document. Such analyses allowed uncovering inconsistencies in the contract-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 specification document, these were documented and modelled as non-deterministic actions in the contract.

The validation of the final contract specification, and indirectly, of the protocol specification document was performed by the experienced engineer. Most of the validation was done by inspection, guided by the enabledness-preserving abstraction (Fig.10) and the modellers expertise, going into the detail of the contract and finally the protocol specification document 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 the protocol specification document:

- In state S6 of the enabledness-preserving FSCA, 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 reply with a continuation token (`rcvInProgess`), which the client cannot accept in state S6. From the document it is not clear what should happen to this continuation token and what should the server side do if it is not accepted.
- States S5, S6 and S7 have outgoing transitions with error labels that go to both the initial and the deadlock state. This non-determinism reflects underspecified behaviour described in the protocol specification document. However, this underspecification seems to be problematic as an implementation of the contract could decide unilaterally whether to (return to the initial state and) try to reuse the connection despite the error or if is 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 enabledness-preserving FSCA for the client constructed automatically from the protocol's contract composed in

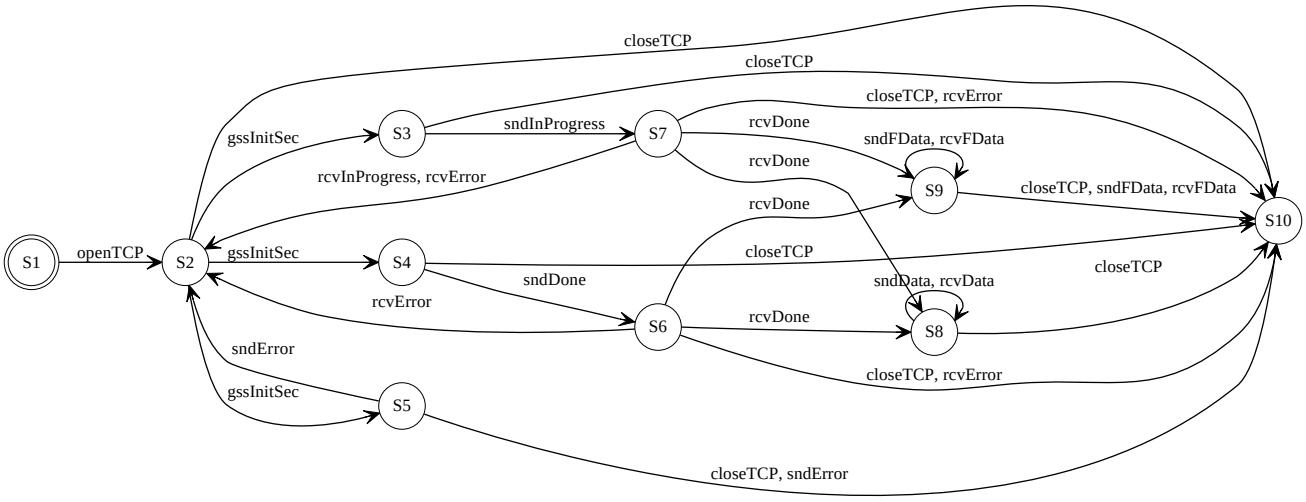

Fig. 10. Enabledness-preserving Finite State Contract Abstraction for the NegotiateStream protocol

parallel with the state machine for the server leads to a deadlock. A trace to the deadlock, raises the following question:

- The FSCA shows that the contract 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 contract 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 specification), hence the issue raised above actually shows an discrepancy between text specification and diagrammatic-aid of the server side in the protocol specification document.

Finally, inspection of the enabledness-preserving FSCA and comparison against the auxiliary client side state machine of the protocol specification documentation helped find some discrepancies between the textual specification and diagrammatic aid for the client side:

- In the state machine of the protocol specification documentation, action `sndError` goes to a state in which the client waits for a message from the server. However, the FSCA that CONTRACTOR produced shows that after this event the client should either terminate the connection or retry the whole phase. The FSCA is consistent with the protocol specification document text.
- Analogously to `sndError`, when the state machine of the protocol specification for the client side receives `rcvError`, the client must wait. However, the FSCA, in agreement with the protocol specification document text, shows that this is not the case.

Various of the issues reported above for version 2.0 of the protocol specification document that was available at the time, were subsequently corrected in version 3.0 of the document. This shows to some extent 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 enabledness-preserving finite state abstraction of an industrial strength document aided significantly in: *i)* correcting and elaborating a formal contract specification of the protocol, in *ii)* identifying relevant problems in the pre-existing real natural language (and auxiliary diagrammatic) protocol specification document, and *iii)* in automatically constructing a diagrammatic aide which is sound with respect to the protocol specification (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 enabledness-preserving FSCA obtained (Figure 10) featured almost the same level of abstraction as the state machines in the protocol specification document. Main differences were that the FSCA 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 artefact 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.

5.5 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 FSCA which has the potential of having billions of states.

We chose another Microsoft protocol specification, in this case the one for the "WINS Replication and Autodiscovery Protocol" [19]. 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 a `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 `nameRecordsRequestControlSuccess`. 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 somehow 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 11 for a diagrammatic representation the process): The initial documentation available was, as with the previous case study, a publicly available protocol specification document [19] including a normative natural language description of the protocol together with diagrammatic aids in the form of state machines, and a SpecExplorer [20] 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, manually, a protocol contract specification that could be input into CONTRACTOR. A systematic translation procedure was used for translating the SpecExplorer model into a contract: a contract variable was created for each of the SpecExplorer model variables and one contract action for each method in the SpecExplorer model. For action preconditions we used the REQUIRES clauses of the SpecExplorer model and for postconditions 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 by 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.

In order to get an FSCA, we proceeded and ran CONTRACTOR on the protocol contract specification obtained by translating the SpecExplorer model. This initial abstraction was produced in about 4 minutes and it had 60 states and 642 transitions. Algorithm scalability, which was the first of our motivations for carrying out this case study, did not seem to be a problem specially

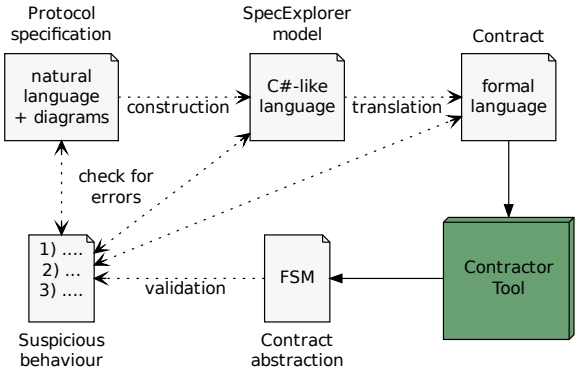

Fig. 11. Experimental setup for the WINSRA case study

considering that we were exploring and producing a result that may have potentially involved 2^{33} (over 8 billion) states.

Regarding the second of our motivations for this case study, feasibility of validating large FSCAs, 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 the MS-NSS protocol case study, an iterative process was enacted in which first the protocol contract was used to produce a FSCA 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 specification document. Errors detected in the contract or in the SpecExplorer model were subsequently corrected and the CONTRACTOR tool executed again.

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

1) The contract 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 FSCA generations, our impression was that there was something wrong with the contract: the natural language protocol description did not seem to be describing a protocol 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 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 contract 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).

action `associationStartRequestControlSuccess()`
pre $\ldots \wedge$ ((association = None $\wedge$ persistent = No) $\vee$ (association = Pull $\wedge$ replicationType = Push) $\vee$ (association = Push $\wedge$ replicationType $\neq$ Push) $\vee$ association $\neq$ Both)
post protocolState' = AssociationStartRequestControl

Fig. 12. Buggy specification for association start request (fragment)

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 contract, we ran CONTRACTOR again 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`. (`initiateTrafficDataVerify` is used for a scheduled data verification process that is similar to the standard pull replication process). 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 contract) involved modifying the REQUIRES clause of the three traffic initiation methods. After this change, the FSCA could be generated in about 3 minutes and had 54 states and 467 transitions.

We further analysed the 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 12. 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 is the protocol currently on (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 association request) the previous action must have been a traffic initiation and the one before that a setup initialization action.

Having corrected this issues in the model and in the contract we ran CONTRACTOR yet again, this time for less than 30 seconds, resulting in an output of 38 states and 233 transitions, which is roughly half the size than before. The following is a list of errors that we found using this FSCA in this iteration:

- The `setupInitialization` action may go to a state with no enabled actions. This was caused by a weak contract invariant, which did not account for the fact that whenever the system was not initialized, 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 reestablish 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 contract 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 contract translation from the SpecExplorer model. In particular, the `updateNotificationObserve` action postcondition may leave a contract 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 FSCA 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 contract specification. However, they do show that validation of FSCAs 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 enabledness-preserving FSCA was helpful in: *i)* correcting and elaborating a formal contract 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 both the FSCA construction algorithm and its implementation can scale to industrial case studies and that the FSCA that is produced, although large, is still amenable to analysis and helpful in finding problems in software development artefacts.

5.6 Case Studies Summary

In this section we have reported on some of the case studies conducted to validate our approach. The case studies provide some evidence regarding the *i)* scalability of the FSCA construction algorithm, *ii)* the size and ease of validation of the resulting abstractions, *iii)* the aid that the abstractions provide in validating and elaborating not only contract specifications but also informal descriptions that can be encoded as contracts, *iv)* the degree to which enabledness-preservation provides an intuitive abstraction level that is close to the level at which developers describe protocols and API expected usage, and *v*) that the spurious behaviour introduced by the abstraction does not hinder the validation process.

Table 1 shows a execution times comparison between the naïve version of the algorithm presented in [12] and the current implementation, which is based on the dependencies analysis presented in Section 4. Not only execution times are much smaller for the case in which both implementations terminate, but also the current implementation is able to cope with the larger case study in less than 4 minutes.

Table 2 shows that reachable states are a very small fraction of the full (and intractable) set of possible states. The amount of reachable states is always larger than $1/3$ of the consistent states. Furthermore, the number of compliant states is the same as consistent states except for the case of the ATM, in which there are 2 inconsistent compliant states. Therefore, the state enumeration algorithm presented in Definition 11 almost always performs just one single query for each consistent state, except

Input		# SMT queries (and running time)		Output	
Name	**# actions**	**Naïf algorithm [12]**	**Current algorithm**	**# states**	**# transitions**
WebFetcher	4	31 (0.15 s.)	35 (0.07 s.)	2	4
ATM	8	535 (3.75 s.)	396 (1.02 s.)	8	30
MS-NSS	13	8454 (72.59 s.)	580 (1.12 s)	10	31
MS-WINSRA (initial)	33	n/a (> 12 hours)	57300 (233.98 s.)	60	642
MS-WINSRA (second)	33	n/a (> 12 hours)	49043 (197.28 s.)	54	467
MS-WINSRA (final)	33	n/a (> 12 hours)	7226 (27.55 s.)	38	233

TABLE 1
Execution times

Name	**Possible states**	**Compliant states**	**Consistent states**	**Reachable states**
WebFetcher	4	2	2	2
ATM	256	10	8	8
MS-NSS	8192	11	10	10
MS-WINSRA (initial)	2^{33}	105	105	60
MS-WINSRA (second)	2^{33}	99	99	54
MS-WINSRA (final)	2^{33}	38	38	38

TABLE 2
Space state sizes

for one case in which it performs 2 extra queries which result in inconsistent states.

The set of compliant states contains all the reachable states in the resulting abstraction and some other inconsistent or unreachable states. These unreachable and inconsistent (yet compliant) states are not very large in number in any of the analysed case studies. In most of the cases there were none of them; in the worst situation this accounted for approximately 50% of the total compliant states. This indicates that the set of compliant states is a good approximation of the set of reachable states.

The best possible construction algorithm would be quadratic in the number of resulting states. Our algorithm is quadratic in the number of compliant states, which is a very good approximation.

6 Validation Guidelines

Based on the experience gained by performing case studies, some of which are discussed in the previous section, we have identified a number of heuristics that aid identification of "suspicious behaviour" during the validation process. This in turn supports identification of problems in the contract specification.

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 enabledness preserving FSCA.

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 modellers can interpret the different states of the enabledness preserving FSCA into the problem domain with relative ease. The first two heuristics we developed confirm, to some extent, this hypothesis.

- **Understanding states**. There are certain abstract states in the enabledness-preserving FSCA that can be easily interpreted to particular situations of the system under analysis. For instance, in the ATM abstraction of Figure 7 the state `S6` is the one which groups those concrete instances on which the card has been inserted but the password has not. In the FSCA of Figure 6 the state `S2` is clearly the one that groups the instances which have an open connection.
  When it is not possible or not easy to associate a particular states with a declarative description of the set of instances that it abstracts, this may be an indication that there is a problem with the specification. 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 FSCA) but that the preconditions of enabled actions or the invariant were (incorrectly) too weak.
- **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 contract. The reviewer should be aware that, given the approximate nature of the abstraction, the appearance of a trace is not a guarantee that it denotes a feasible action sequence.

We also identified the following *structural characteristics of an enabledness-preserving FSCA* that can help pinpoint problems in a contract specification:

- **Large state space**. A large state space in the enabledness-preserving FSCA may be an indication of either a poorly designed set of operations or an incorrectly specified contract. 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 class or an 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 incorrectly specified. More specifically, a large state space can be an indication of problems with preconditions. A good strategy is to question why different states in the enabledness-preserving FSCA differ in the actions that they enable.

- **Deadlock states**. The presence (or absence) of a deadlock state is something that should be analysed in detail when validating a contract specification using enabledness-preserving FSCAs. By definition of enabledness-preserving FSCA there can be only one deadlock state, the state whose action set is empty.
  The presence of an unintended deadlock state in an FSCA is likely to be an indication of a problem with a condition that is stronger than needed. Particularly, a problem with a postcondition (an operation that leads to a state where no preconditions hold), with an invariant (which in conjunction with a postcondition makes all preconditions infeasible), or with a precondition (that disallows an operation that would prevent the deadlock)
- **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.
- **Missing action**. If a given specified action is not present in any of the enabledness-preserving FSCA reachable states then this is an indication that something is not quite right. It may be the case that the precondition for that action is inconsistent when combined with the contract invariant. It might also be the case that none of the other actions' postconditions leave the system in a state which enables the missing action.
- **Enabled action with missing transitions**. If a state which enables a certain action does not have any outgoing transition labelled with that action name, then the postcondition for that action is a contradiction.
- **High fan-in**. States in an enabledness-preserving FSCA 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 preconditions that when corrected end up partitioning the high fan-in state into several states.
- **Highly non-deterministic actions**. When a state has a large number of outgoing transitions labelled 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 postcondition for the action is non-deterministic. This can occur for instance if the postcondition underspecifies the behaviour of the operation; underspecification which can be intentional or not. The latter case requires strengthening the postcondition for the action. Secondly, a highly non-deterministic action on a state can also happen if the invariant for the state is weak. For instance, an action with a postcondition of the form $(A_1 \Rightarrow B_1) \wedge \ldots (A_n \Rightarrow B_n)$ may generate undesired non-deterministic behaviour in a state where several A_i hold. In these cases, it may be the case that a precondition or the contract invariant requires strengthening.
- **Mirrored actions**. If whenever there is a transition labelled with a given action a_1, there is another transition with the same origin and destination state labelled 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 system designer forgot to modify the appropriate differences between the two (known as copy-paste bugs).

Finally, if the contract is modelling a reactive system (such as the MS-NSS and MS-WINSRA protocols), then actions are categorised as either controllable or monitorable. In this context, the following strategies can also be applied when validating via enabledness-preserving FSCAs:

- **Connected subsets with no monitorable actions enabled**. If a connected subset in the enabledness-preserving FSCA does not include any state that accepts monitorable actions, this is indicative of a problem: The specification allows blocking the environment or at the very least fails to consume relevant events controlled by the environment.
- **Mixed controllable and monitorable enabled actions**. If a state contains a mixture of controllable and monitorable actions this is indicating that the system is ready to both engage in an operation on its own or receive a stimulus from its counterpart. This could be considered highly suspicious in the context of a contract describing a synchronous system.

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

7 Discussion and Related Work

The use of abstraction to address the complexity of contracts introduces spurious behaviour in the same way abstraction for verification does. This can hinder

both verification and validation if too much spurious behaviour is present in the abstraction. Automated refinement techniques can be applied in abstract verification methodologies to resolve this problem: When a spurious counter-example is identified, the abstraction can be refined, with some semantic preservation criterion, to remove the spurious behaviour (e.g., [21]). Although a scheme such as this one could be replicated in the context of abstraction for validation such as ours, the resulting refined model will most likely cease to be tractable for validation rapidly. The case studies we conducted have shown that although the enabledness-preserving abstraction is rather coarse grained, and hence it introduces spurious behaviour, it still supports finding real errors in real specifications.

The notion of abstraction used in this paper relates to that of a finite state machine simulating all possible implementations of a contract. An alternative approach to the one presented here is to define a notion of canonical implementation of a contract and then apply some well-known abstraction techniques over that possibly very large or even infinite state space. Such an approach would allow analysis of the abstraction to provide stronger guarantees on possible contract implementations. However, even being a sensible approach for verification, validation of such a model would be significantly hindered in our opinion: Firstly, requiring preservation of behaviour implies that a minimal finite state abstraction may not exist. For instance, the circular buffer example used in Section 2 would not have a finite state bisimulation abstraction. Secondly, even if a finite bisimulation exists, the size of the abstraction may be too large to validate. In fact, in the case of the NegotiateStream protocol would have a bisimulation abstraction that is roughly twice the size than ours since it would have to account for the requested protection level from the first call to GSSinitsec operation done by the client. Finally, unlike our approach, given an abstract state, predicates characterising which concrete states are represented by it (i.e., "abs^{-1}") are likely to be cumbersome and hard to relate with the original contract predicates.

The work presented in this paper extends previous results presented by the authors ([12]). More specifically, we now present a novel FSCA construction algorithm that scales to industrial strength case studies, discuss the CONTRACTOR tool which supports the approach, reports on a large case study that features a contract with 33 actions. The size of this case study was beyond that which could be addressed with the algorithm presented in [12], and discuss lessons learnt based on the validation of our approach providing, in particular, some heuristics that were used to identify errors in the industrial case studies we conducted.

Predicate abstraction

Our work can be considered as instantiating the framework of predicate abstraction [8], [21] in that we produce abstractions based on predicates that characterise the enabledness of sets of operations. While resting in this framework, the contributions of this work are the selected level of abstraction and its application to the problem of contract validation, together with an algorithm and validation guidelines.

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) [22]. Although there are similarities with our work in how transitions are computed the key difference is in the predicates used for abstraction: In [22] 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 (CEGAR) sometimes need an initial model and a property from which then the iterative process is performed. In fact, we believe that FSCAs may serve that first purpose.

Techniques that construct FSMs from declarative requirements specifications [6] 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.

A level of abstraction somewhat related to that of enabledness has been used in [23]. The authors quotient the state space of a class based on the result of its parameterless boolean observers. The abstraction is used for test-case generation aiming to obtain state coverage of the abstraction. 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. By fixing the abstraction level using the boolean observers, the resulting finite state machine is heavily dependant on the quantity and quality of these observers.

Our technique is related with [24], meant for checking invariants, which builds an abstract state graph out of a guarded transition system and a set of input predicates. Concrete states are abstracted by using a sub-lattice of monomials of abstract boolean variables representing the truth values of the input predicates. Notice that an approach based on monomials would yield a 3-valued denotation: an action is enabled (positive literal in the monomial), an action is disabled (negative literal in the

monomial) or an action may be enabled (when the literal is absent in a monomial). In contrast, our technique defines states by formulas determining the enabledness of each action precondition (the atoms of the lattice).

Even setting the input predicates in [24] to model the enabledness conditions of actions, their level of abstraction would generate less intuitive states. For instance, in the case of the circular buffer, partially full buffers would be mapped to 3 different abstract states. Furthermore, there would not be any abstract state uniquely characterising the full circular buffer. Their level of abstraction might also generate larger state spaces (up to 3^n).

In [24] transitions are determinised by using the least monomial among all successors in the concrete system. On the other hand, we allow non-determinism in transitions, which results in a more compact state machine, which we believe could be practical for validation.

Contract exploration

Other contract validating techniques such as the ones presented in [16], [25], [26] 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, in a complementary manner, the validation of contracts.

The ideas presented in [2] are also aimed at validation of contracts 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.

LTS minimisation

The comparison with behavioural abstractions is linked to the minimisation problem of transition systems [27], [28]. These algorithms are based on finding a maximum fixed point by stabilising state space partitions. Besides the shortcomings mentioned regarding requiring bisimilarity in our setting, in general, such 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 [28] 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 [28], 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).

Typestates and interface learning

Our technique is related to approaches that synthesize typestates [29], [30] or interfaces [31], [32], [33] from 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 to obtain a set of safe traces from a client perspective (every trace in the abstraction must be accepted by the original program), using abstraction for verification purposes rather than validation.

For instance, in [31] this safety requirement tends to make their abstractions overly restrictive in terms of the model behaviour. This particular work aims at creating finite models out of Java classes with the focus in client code, trying to safely capture as much environmental behaviour as possible while assuring that the program never throws an exception on any model path. This restriction make their approach unfit for validation purposes since limiting the model to safe behaviour may render a trivial state machine.

Consider the circular buffer example in Figure 1. The non-regular nature of the underlying language makes the approach in [31] render a finite state machine like ours but without the loop transitions in the middle state. This is due to the fact that the only behaviour for a circular buffer of size greater or equal to 2 that is always available is writing at most twice and then reading. Our technique does not guarantee that any path is exception-free on any implementation of the contract. By not trying to enforce this condition, we are able to produce finite state machines that aid validation by showing as much behaviour as possible. The restriction due to client safety can be clearly seen in the evaluation section of [31] in the interface of Figure 6 where the authors limits the number of observed exceptions in order to show a more meaningful state machine. Furthermore, tracing back the interface elements to the original artefact is not an easy task using [31] since states are constructed as needed by the learning algorithm. On the other hand, our technique favours validation by having a direct relationship between states and enabledness of actions.

While keeping the safety requirement, approaches such as [32], [33] successfully deal with the problem of inferring more permissive interfaces. They do so at the cost of assuming certain conditions over the artefacts they analyse, for instance the algorithm in [32] requires the software component to be described by a finite automaton. The algorithm in [33], as the one in [31], construct abstractions which are safe with respect to the throwing of a single exception. Even under these limitations, obtained abstractions are potentially less permissive than ours due to the safety requirement. Furthermore, tracing the abstraction back to the original

artefact is not necessarily easy with their abstractions since there is not a clear relation between the obtained model and the elements of the artefact under analysis.

Behaviour model mining

Dynamic invariant detecting tools such as Daikon [34] have proven useful in many contexts. In our case, Daikon could be used to obtain pre/post conditions, as well as invariants, for a particular program. With this information, we could proceed and create a FSCA for that program and produce a graphical and concise representation of the extensive amount of information that Daikon produces. Using this configuration we would be able to provide the user an on-line abstraction of the program he is writing. We would have to take into account that the assertions that Daikon outputs are true for the runs that it used to create them, yet not necessarily true for all the possible runs.

Our approach relates to the mining of temporal specifications (e.g., [35]), which aims at producing, from traces, a finite state automata that describes how a set of operations is used. The main difference with our work 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 a contract. In addition, mining techniques have a dynamic flavour and their results heavily depend on the quality of the traces used as input. On the other hand, our technique *statically* yields a model that is an abstraction of any legal implementation of a given contract.

Furthermore, behaviour model mining tools such as ADABU [36] produce finite state machines whose states are determined by a fixed abstraction over the return values of all the inspectors in a class. For instance, integers are abstracted according to its sign, therefore this technique is not suitable for differencing two significant states defined 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 our abstraction is going to consider them by means of the SMT solvers.

Run-time behaviour observation is also used in [37], in which the authors provide a way to generalise component behaviour using samples taken during a systematic bounded execution. In a first step a deterministic finite state machine is built using the sampled behaviour. This is then generalised via graph transformation rules and invariant detection tools. If a canonical contract implementation were to be sampled using this technique then we would end up having a set of graph rules tightly correlated to the original contract. That is, the technique would traverse the inverse path we define in our work.

A similar approach can be found in [38], 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. This technique is similar to the previous one in the sense that it (unsoundly) generalises observed behaviour by applying invariant detecting tools. Also, unlike our approach, the amount and quality of behaviour space synthesized depends on the traces used as input. On the other hand, there is no clear indication that yielded abstractions would be coarse enough for validation.

8 Future Work

The problem of supporting the analysis of FSCAs is a line of work that we aim to study. For instance, data and control slicing of contracts and FSCAs may provide useful tools for designers, however these operations need to be studied formally, property preservation results obtained and efficient algorithms developed. Of course, relatively standard tool support such as compacting the FSCA using hierarchical states like those in UML statecharts or decomposition into communicating FSCAs may be useful.

In addition, there is the issue of model precision, the algorithm presented in this paper introduces, as any abstraction, spurious paths. Can a more precise enabledness preserving abstraction of a contract be produced? One potential direction is to introduce modalities into our finite state abstractions to distinguish between may transitions (FSCAs current interpretation) and must transitions that can always be traversed by selecting the appropriate parameter values. Another potential direction is to analyse the effect of postconditions allowing to produce stronger state-invariants, such effects could be propagated using a fix-point algorithm.

Finally, although a number of industrial case studies have been performed these have focused on software protocols, application of the approach to other kinds of systems may yield more refined heuristics for detecting problems in contract specifications.

9 Conclusion

We conjecture that contract validation would benefit from easily auditable abstractions that exhibit global implications of locally specified behaviour. In this paper, we have showed the potential validation capacity of enabledness-based contract abstractions. We have provided a novel symbolic algorithm that leverages the concept of action enabledness dependencies to efficiently construct finite contract abstractions that are both concise and handy for validation purposes. We have implemented our algorithm as a practical tool and used it to get finite abstractions of a variety of contracts. These finite models led us to discover previously unknown inconsistencies or omissions in real-life specifications. We have provided a set of guidelines that we believe are useful when validating contract specifications by means of enabledness-preserving abstractions. Finally,

we believe that the succinctness of the abstractions we obtain with our technique makes them valuable and versatile tools when constructing or analysing contracts.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their time and insightful comments which have greatly improved this paper. The work reported herein was partially supported by CONICET, UBACyT X021, ANPCyT PICT 32440, PIP112-200801-00955KA4, and PICT-PAE 37279.

REFERENCES

[1] N. Polikarpova, I. Ciupa, and B. Meyer, "A comparative study of programmer-written and automatically inferred contracts," in *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*. New York, New York, USA: ACM Press, 2009, p. 93.

[2] H. Van, A. van Lamsweerde, P. Massonet, and C. Ponsard, "Goal-oriented requirements animation," in *Requirements Engineering Conference, 2004.*, 2004, pp. 218–228.

[3] B. Meyer, "Applying design by contract," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[4] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, 1995.

[5] W. Grieskamp, N. Kicillof, and N. Tillmann, "Action machines: a framework for encoding and composing partial behaviors," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 5, pp. 705–726, 2006.

[6] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Deriving event-based transition systems from goal-oriented requirements models," *Automated Software Engineering Journal*, vol. 15, no. 2, pp. 175–206, 2008.

[7] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios," in *ICSE'00*, 2000, pp. 314–323.

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

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

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

[11] J. Esparza, "Decidability of model checking for infinite-state concurrent systems," *Acta Informatica*, vol. 34, pp. 85–107, 1997.

[12] G. de Caso, V. A. Braberman, D. Garbervetsky, and S. Uchitel, "Validation of contracts using enabledness preserving finite state abstractions," in *ICSE*. IEEE, 2009, pp. 452–462.

[13] 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)*, Jul. 2004, boston, Massachusetts.

[14] D. Harel, "StateCharts: A Visual Formalism for Complex Systems," *Science of Comp. Program.*, vol. 8, pp. 231–274, 1987.

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

[16] W. Grieskamp, N. Kicillof, D. MacDonald, A. Nandan, K. Stobie, and F. L. Wurden, "Model-based quality assurance of Windows protocol documentation," in *ICST*. IEEE Computer Society, 2008, pp. 502–506.

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

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

[19] "[MS-WINSRA]: Windows Internet Naming Service (WINS) Replication and Autodiscovery Protocol Specification," May 2009, http://msdn.microsoft.com/en-us/library/dd304175(PROT.13).aspx.

[20] 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.

[21] T. Ball and S. Rajamani, "The SLAM project: debugging system software via static analysis," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002.

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

[23] L. Liu, B. Meyer, and B. Schoeller, "Using Contracts and Boolean Queries to Improve the Quality of Automatic Test Generation," *Lecture Notes in Computer Science*, vol. 4454, p. 114, 2007.

[24] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *CAV '97*, 1997, pp. 72–83.

[25] M. Leuschel and M. Butler, "ProB: an automated analysis toolset for the B method," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 10, no. 2, pp. 185–203, 2008.

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

[27] D. Lee and M. Yannakakis, "Online minimization of transition systems (extended abstract)," *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pp. 264–274, 1992.

[28] S. Tripakis and S. Yovine, "Analysis of Timed Systems Using Time-Abstracting Bisimulations," *Formal Methods in System Design*, vol. 18, no. 1, pp. 25–68, 2001.

[29] R. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. 12, no. 1, pp. 157–171, 1986.

[30] R. DeLine and M. Fahndrich, "Enforcing high-level protocols in low-level software," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. ACM New York, NY, USA, 2001, pp. 59–69.

[31] R. Alur, P. Černỳ, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for Java classes," *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 98–109, 2005.

[32] D. Giannakopoulou and C. Păsăreanu, "Interface Generation and Compositional Verification in JavaPathfinder," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. Springer, 2009, pp. 94–108.

[33] T. Henzinger, R. Jhala, and R. Majumdar, "Permissive interfaces," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM New York, NY, USA, 2005, pp. 31–40.

[34] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, 2007.

[35] M. Gabel and Z. Su, "Symbolic mining of temporal specifications," *Proceedings of the 13th international conference on Software engineering*, pp. 51–60, 2008.

[36] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *International Conference on Software Engineering: Proceedings of the 2006 international workshop on Dynamic systems analysis*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 2006.

[37] C. Ghezzi, A. Mocci, and M. Monga, "Synthesizing intensional behavior models by graph transformation," in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering-Volume 00*. IEEE Computer Society Washington, DC, USA, 2009, pp. 430–440.

[38] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th international conference on Software engineering*. ACM New York, NY, USA, 2008, pp. 501–510.

Guido de Caso is a doctoral candidate in computer science at the Department of Computing, FCEyN, Universidad de Buenos Aires, where he is also a teaching assistant in charge of undergraduate software engineering courses practical sessions. His research interests include contract validation, software verification, program analysis and programming languages design. He has been awarded a graduate fellowship by Microsoft Research.

Víctor Braberman holds a Professorship at the Department of Computing, FCEyN, Universidad de Buenos Aires and he is a CONICET researcher working in the area of Software Engineering. His is an active researcher in models, abstractions and verification of software intensive systems. He has participated in several European projects and he was visitor researcher at several institutes: Politecnico di Milano (Italy), VERIMAG (France), UNU/IIST (Macao) and Imperial College (UK). He has headed the development of tools for the analysis of real-time systems. He has a varied industrial national and international experience as an independent consultant on SE and testing topics. He was chair of ASSE 2003 (Argentine Symposium on Software Engineering) and served as PC member of ICSE, FSE, ISSTA, FASE, ASE and ICTAC as well as ICSE workshops PC co-chair. He has also been awarded in several occasions by Microsoft Research and IBM Eclipse Innovation program.

Diego Garbervetsky holds a Professorship at the Department of Computing, FCEyN, Universidad de Buenos Aires and he is a CONICET researcher working in the area of Software Engineering. His research interests are in software verification and program analysis, in particular contract verification and inference of quantitative information about dynamic memory utilisation for object oriented programs. He has participated in several European projects and also been awarded by Microsoft Research and IBM Eclipse Innovation program.

Sebastián Uchitel holds a Professorship at the Department of Computing, FCEyN, Universidad de Buenos Aires, he is a CONICET researcher and has a Readership at the Department of Computing, Imperial College, London. His research interests are in behaviour modelling and analysis of requirements and design for complex software-intensive systems. His research focuses on partial behaviour modelling, including scenario-based specifications, behaviour model synthesis and modal transition systems. Dr. Uchitel was associate editor of IEEE Transactions on Software Engineering and is on the editorial board of the Requirements Engineering Journal, he was program co-chair of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006) held in Tokyo and is the program co-chair of the 32nd IEEE/ACM International Conference on Software Engineering (ICSE 2010).