Validation of Contracts using Enabledness Preserving Finite State Abstractions

Guido de Caso* Víctor Braberman* Diego Garbervetsky* Sebastián Uchitel*†

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

† Department of Computing, Imperial College
London, UK
su2@doc.ic.ac.uk

Abstract

Pre/post condition-based specifications are commonplace 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. The level of abstraction at which such models are constructed preserves enabledness of sets of operations, resulting 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 an industrial strength protocol specification in which concerns were identified.

1. Introduction

Pre/post condition specifications constitute good practice in a variety of software engineering activities [16]. In requirements engineering they provide the link between declarative high-level system goals, and operational requirements for the software-to-be [21]. Use case specifications, which are popular in development processes such as RUP are also equipped with pre and postconditions. In design, the notion of design by contract [14] is underpinned with pre/post conditions. At the code level, the use of assertions to verify at run-time pre/post conditions is considered good-practice [17].

A pair pre/post condition 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. Although writing the pre/post condition for an operation requires expertise, it is a comparably simple task compared with writing pre/post conditions for a set of related operations. Ensuring that the pre/post conditions of a set of operations of an API are correct requires a cohesive model of how the various pre/post conditions should fit together. 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.

Behaviour models such as finite state machines and action machines [9] 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 [11], use cases, and scenarios [22]. 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 artifacts that rely on pre/post conditions as a means for specification by automated construction and tool-supported analysis of behaviour models.*

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 cap-

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

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 $rv = a[rp'] \ \wedge \ wp' = wp \ \wedge \ a' = a$
$\wedge \ (rp < |a| - 1 \ \Rightarrow \ rp' = rp + 1)$
$\wedge \ (rp = |a| - 1 \ \Rightarrow \ rp' = 0)$

Figure 1. Specification of a circular buffer

turing 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 models constructed by the approach described herein can be used to validate contract pre/post-condition-based specifications through inspection, animation, simulation, and model checking. 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 represents an invariant expressed 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).

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) a prototype implementation that constructs enabledness preserving abstractions from contracts, and iv) the validation of an industrial strength protocol specification in which our approach supported the identification of a number of ambiguities and inconsistencies.

The rest of this paper is organised as follows. We begin with a motivational 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 a prototype implementation of our approach and the validation of the tool and approach on three case studies (Section 5). Finally, we discuss related work (Section 6), ideas for future work (Section 7) and conclusions (Section 8).

2. Motivation

In this section we illustrate the difficulties of validating pre/post-condition specifications using a toy example in order to motivate our approach.

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 size, 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 and requires both pointers to be within the bounds of the circular buffer, i.e. between 0 and $size$, 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 our mental model of what a circular buffer is? A traditional approach is to establish relevant declarative properties that should be satisfied by the specification. For instance, we could postulate:

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.

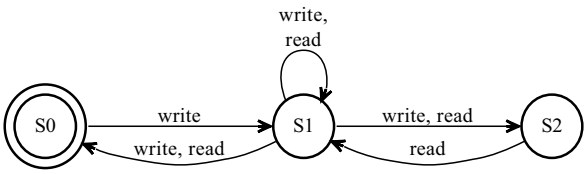

Figure 2. Circular buffer finite abstraction

ExtendedCircularBuffer

$\vdots$

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

$\vdots$

action `reset()`

pre true

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

Figure 3. Circular buffer with `reset`

The process of verifying these properties has a number of difficulties. Properties must be formalised and a formal reasoning framework which can accommodate the properties together with the specification must be identified. Having done this, the actual reasoning to establish correctness of the specification with respect to the properties is complex because it involves combining the various elements present in the specification such as the pre and postconditions of different actions, invariants and initial states. For example, to informally prove that the first property holds we must use the initial state condition to infer that $wp = 0 \ \wedge \ rp = |a| - 1$, which makes the precondition of `write` valid. Then we must use the postcondition of `write` to show that $wp = 1$ after it's execution. Finally we have to prove that $wp = 1 \ \wedge \ rp = |a| - 1$ is enough to force the precondition of the `read` action. Even after the non-trivial process of proving all of the desirable properties that we can think of, the question of whether the set of properties used for verification is correct and complete remains. Have we formalised the properties accurately? Have we included all the relevant properties?

We believe that automated construction of abstractions that consolidate pre/post condition specifications into one cohesive behaviour model can complement existing techniques providing support for further analysis and validation of pre/post specifications. Consider, for instance, the behaviour model shown in Figure 2 of the circular buffer specification which abstracts away the size of the buffer and brings an infinite state space down to only three states. Transitions between states show the applicability of circular buffer actions depending on its state. The behaviour model allows an engineer to validate in a very simple way the specification against his or her mental model of the circular buffer. The model conveys very clearly that there are three relevant abstract states of circular buffers which relate to whether the buffer is empty, full, or neither: 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.

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`
- State 1: `inv` $\wedge$ `write_pre` $\wedge$ `read_pre`
- State 2: `inv` $\wedge$ `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` can lead to state 0, question which leads `write`'s postcondition. 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 0, 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 without the two suspicious transitions we described. 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 with legal operations (those that preserve the incomplete invariant) such as the specification shown in Figure 3.

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, the specific choice of level of abstraction of the model, and the traceability of the abstraction to the specification helps identifying and fixing problems in the latter. 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 3.

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*[1]*.*
- $Q : A \to \mathbb{P}(V \cup V' \cup \{p\})$ *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.*

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

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

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

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 [9]). Data State Machines have states labelled by mappings from variable names to a given value domain while transitions are labelled with actions together 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 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* $s \in \mathcal{S}_0$*.*
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 its elements* s' *are all included in* $\mathcal{S}_v$*. Furthermore,* $Q_{a_i}(s \cup s' \cup \{p \mapsto d\})$ *holds*[2]*.*

[1] 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.

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

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

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 booleans 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 $abs_I : \mathcal{S}_v \to S$ 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.

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.

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.*

In order to construct an enabledness-preserving FSCA we first need to define 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:*

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

Using action set invariants the construction of an enabledness-preserving abstraction is straightforward:

Definition 8 (Enabledness-preserving FSCA Construction Algorithm)**.** *Given a contract* $C = \langle V,\ inv,\ init,\ A,\ P,\ Q\rangle$ *and an interpretation* $\langle \mathbb{D}, \mathbb{D}^{op}\rangle$*, we proceed to build an FSM* $M = \langle S,\ S_0,\ \Sigma,\ \delta\rangle$ *where*

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

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

Observe that the constructed enabledness-preserving FSCA never has two states with the same available actions.

It is straightforward to prove that the FSCA constructed according to Definition 8 is an enabledness-preserving abstraction.

Theorem 1. *Given a contract* C *and an interpretation* $\langle \mathbb{D}, \mathbb{D}^{op}\rangle$*, then* M *as built by Definition 8 is an enabledness-preserving FSCA of* C *under the* $\langle \mathbb{D}, \mathbb{D}^{op}\rangle$.

The proof for the theorem can be simply 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 such that every pair of concrete states s, t satisfy that $s \equiv_e t \Leftrightarrow abs_I(s) = abs_I(t)$.

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 Definition 8 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.

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.

5.1. Tool Support

In order to validate our approach we built a prototype tool[3] in PYTHON that takes a contract description as input and returns an enabledness-preserving finite state contract abstraction. It uses Satisfiability Modulo Theories (SMT) solvers [4] to reason about satisfiability of the formulae as described in Section 4. In cases where these solvers timeout 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 Definition 8 allows these conservative decisions, and Theorem 1 guarantees the construction of a proper abstraction of the contract. In any case, for the case studies discussed in this section, no transitions were added as a result of limitations of the SMT solvers.

The rest of this section deals with three case studies used to test the tool, its capabilities and, more importantly, to validate the approach. Note that even this initial and naive prototype implementation is capable of dealing with reasonably complex examples in times that range from a couple of seconds up to about 2 minutes in a standard desktop computer (Core2Duo with 2GB of RAM memory).

5.2. .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.

Inspired by quality process and model-based testing approach described in [8] we selected as case study subject a Microsoft protocol specification currently under revision:

[3] The tool is available online for download together with the case studies used in this work at `http://lafhis.dc.uba.ar/contractor`.

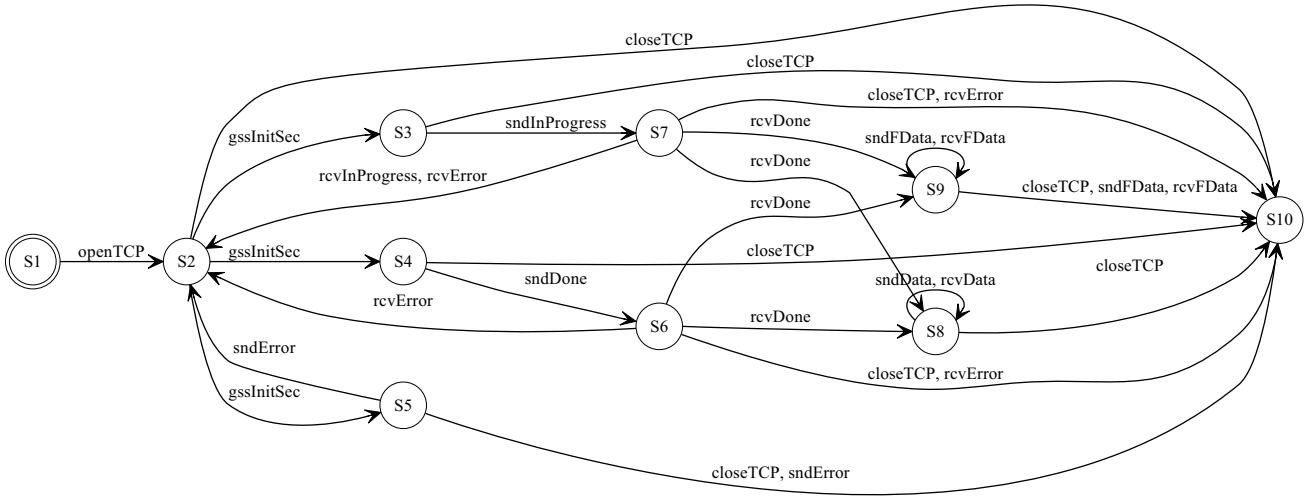

Figure 4. Enabledness-preserving Finite State Contract Abstraction for the NegotiateStream protocol

The MS-NSS protocol [1] 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 [13] 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 negotiation phase, data exchange phase can result in an error in which case the communication is usually terminated.

The case study was conducted 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 ([1][4]). Then, the same person wrote a contract for the protocol validating the protocol against the document and using as sole automated support the prototype described above. Once the protocol's contract was completed, an engineer with experience in protocol validation analysed the enabledness preserving abstraction produced by our prototype in order to validate the contract specification and the protocol specification document itself.

The protocol specification document is structured natural language description containing two auxiliary state machines. The natural language description is considered the normative specification of the protocol, the state machines are simply aides and references for the reader.

The protocol contract developed was actually conceived as a set of controllable and observable actions appearing in the specification of client side. Both natural language and auxiliary state machines were used to aid the comprehension of the protocol specification document, but only the information provided in natural language was used as a source for the contract-to-be. It is worth mentioning that models developed in [1] include server-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 not resort to server-side specification section of the document.

During the contract development process the modeller used the enabledness preserving FSCA produced by our tool to eliminate bugs and typos from the specification being

[4]Version 2.0 was the most updated version when this paper was first sent for review. Since then, version 3.0 was published, which corrects some of the inconsistencies we describe.

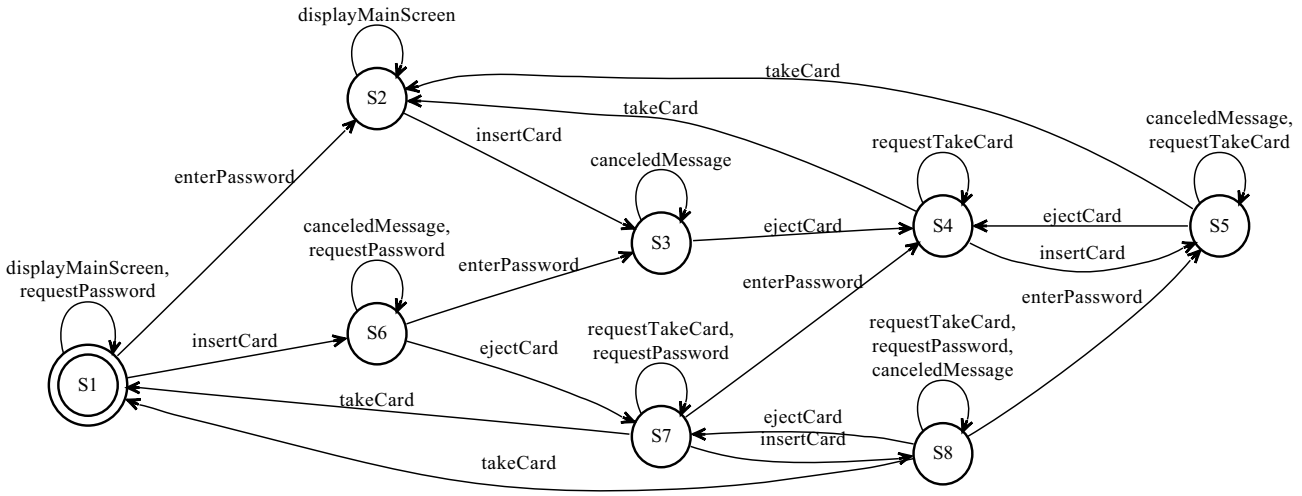

Figure 5. Enabledness-preserving Finite State Contract Abstraction for the ATM

developed. The FSCA was analysed using inspection techniques and simulating scenarios appearing in 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 modeled 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 enabledness preserving abstraction (Fig.4) and the modellers expertise, going into the detail of the contract and finally the protocol specification document if needed.

As a result of the validation three kinds of issues were raised. First, two questions regarding the behaviour of the client were raised as a result of identifying some aspects that were unclear in the protocol specification document:

- A client may receive a message `rcvInProgess` after sending message `sndDone`? This means that the client determined that the negotiation phase is over whereas the server acts differently.
- In case of error, should the client close the TCP connection and start from scratch or should it retry using the same connection?

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

- May a client receive `rcvDone` without ever sending a `sndDone` message? This implies that the server may unilaterally decide to enter the data transfer phase.

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-aide of the server side in the protocol specification document.

Finally, inspection of the enabledness-preserving FSCA also helped find some discrepancies between the textual specification and diagrammatic aide for the client side:

- In the state machine `sndError` goes to a state in which they wait for a message from the server. According to the protocol specification document after this event the client should either terminate the connection or retry the whole phase; none of these situations involves waiting for messages. The FSCA our tool produced is consistent with the protocol specification document text.
- An analogous situation happens with `rcvError`.

It is worth mentioning that the enabledness-preserving FSCA obtained (Figure 4) featured almost the same level of abstraction than the state machines in the document[5]. We find this observation interesting because we believe that hand-written diagrammatic-aides although useful to convey information are usually inaccurate or out-of-date artefacts.

[5]Our FSCA has more states and transitions because contract models local GSS-API calls explicitly and we distinguish encrypted and plain data transmissions.

In summary, we believe the automated construction of an enabledness preserving finite state abstraction of the protocol's contract aided in the development of a pre/post-condition based specification of an existing industrial strength protocol and also supported identifying potential issues with the protocol's existing documentation.

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

Figure 6. Specification of a web page fetcher

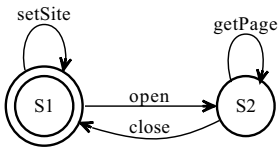

Figure 7. FSCA for the web page fetcher

5.3. ATM

We applied our approach to the ATM case study. To validate our results we used the work described in [22] where a StateChart model is inferred from scenarios and pre/post conditions for actions appearing in them.

We fed our tool with the pre/post specification used in [22] to construct a StateChart for the ATM and obtained the enabledness-preserving FSCA in Figure 5. Note that we did not use the scenarios provided in [22] .

We compared our FSCA model with the StateChart provided in Figure 11 of [22] with respect to simulation. 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 compared the invariants of the states reached by the execution up to `takeCard` and found that they differed on the acceptable values for the $passwdGiven$ system variable. It turns out that `takeCard`'s postcondition does not update the $passwdGiven$ system variable to false. The impact of this is that, according to the pre/post specification in [22], the ATM never returns to a state where it can accept a new password to be entered because it already has one.

In summary, the construction of an enabledness-preserving FSCA from the pre/post specification of an ATM in [22] supported uncovering errors in the specification and also shows that the scenario synthesis technique therein proposed does not preserve postconditions of operations.

5.4. WebFetcher

We considered a case study presented in [5] 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 seen in Figure 6.

Our tool applied to the web fetcher contract results in the FSCA depicted in Figure 7. 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 [5]. Furthermore, our technique could be extended to produce typestates automatically from contracts.

6. Discussion and Related Work

The notion of abstraction used in this paper relates a finite state machine with all possible implementations of a contract. An alternative approach is to define one canonical implementation of a contract and then have a stronger notion of abstraction that preserves simulation equivalence or even bisimilarity with respect to that implementation. Such an approach would allow analysis of the abstraction to provide stronger guarantees on the implementation; However, validation of such a model would be significantly hindered: 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. For instance, 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, 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. Note that an enabledness preserving abstraction can be viewed as "1-step bisimulation" abstraction.

Our work can be considered as part of the broad area of predicate abstraction [20, 3] in that we produce abstractions based on predicates that characterise the enabledness of sets of operations. Within this area, a closely related technique is the construction of finite state machines from Z specifications (which include pre and postconditions) and Live Sequence Charts (LSCs) [18]. Although there are similarities with our work in how transitions are computed the key difference is in the predicates used for abstraction: In [18] 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 from which then the iterative process is performed. We believe that FSCAs may serve that purpose.

Other contract validating techniques such as the ones presented in [8, 12, 15] explore the state space of a given contract either symbollicaly 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 [21] 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 a bounding the number and values of propositions and predicates.

The comparison with behavioural abstractions is linked to the minimisation problem of transition systems [10, 19]. 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 [19] 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 [19], 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).

Dynamic invariant detecting tools such as Daikon [6] 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 online 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 non necessarily true for all the possible runs. Other techniques, such as [2], aim at creating finite models out of programs via predicate abstraction; our work differs in that we are interested in generating abstractions directly from the contract, regardless of any program that may implement it.

Our approach relates to the mining of temporal specifications (e.g. [7]), 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 side 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.

Techniques that construct FSMs from declarative requirements specifications [11] 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 the 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 than the specification.

7. Future Work

The scalability of our tool remains an issue. Complexity is exponential on the number of actions (the number of abstract states is in the worst case the powerset of actions A), however evidence shows that the enabledness-equivalence partition of states is much smaller than $2^{|A|}$. Hence, an on-the-fly exploration of the state space could be a way of drastically improving scalability. Parallelisation of the algorithm is also possible, since the expensive transition discovery process is independent from one abstract state to an-

other. Nonetheless, as reported in the Case Studies Section, performance for a industrial sized protocol was not an issue.

Although in the discussion we mention that aiming for property preservation, such as in bisimulation abstractions, hinders practical validation, we believe that extending our abstractions with modalities might be a way to get small abstractions that provide more property preservation. More concretely, introducing modalities into our finite state abstractions could help distinguish between may transitions (FSCAs current interpretation) and transitions that can always be traversed selecting the right parameter values.

Finally, we would like to introduce the possibility to extend FSCAs with hierarchical states such as in UML Statecharts by considering abstractions that arise from the omission of subsets of actions. Another interesting feature would be to allow the decomposition of the FSCA into communicating FSCAs that run in parallel. Both features may provide more compact representations that allow a better visualisation the different concerns involved in a contract.

8. Conclusions

In this paper, we have precisely formalised the concept of finite behavioural contract abstractions, showing their potential validation capacity. We have provided a novel symbolic algorithm that leverages the concept of action set enabledness to get a finite contract abstraction that is 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.

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.

9. Acknowledgements

The work reported herein was partially supported by CONICET, ERC 204853/PBM, UBACyT X021, and ANPCyT PICT 32440.

References

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

[2] 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*, pages 98–109, 2005.

[3] 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.

[4] 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)*, July 2004. Boston, Massachusetts.

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

[6] 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.

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

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

[9] 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*, 16(5):705–726, 2006.

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

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

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

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

[14] B. Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.

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

[16] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. *month*, 9(608):11, 2008.

[17] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

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

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

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

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

[22] J. Whittle and J. Schumann. "Generating Statechart Designs from Scenarios". In *ICSE'00*, pages 314–323, 2000.