Simulation of contracts using enabledness-preserving Finite State Abstractions
Simulación de contratos mediante abstracciones de comportamiento basadas en habilitación

Alexis Tcach

LU: 219/00

Departamento de Computación, FCEyN, UBA
Buenos Aires, Argentina
atcach@dc.uba.ar

Directores:
Guido de Caso
Diego Garbervetsky

August 10, 2010

Abstract

Pre/postcondition-based specifications are commonplace in a variety of software engineering activities that range from requirements 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 previous work, automatically constructed abstractions in form of behavior models synthesized from pre/postcondition-based specifications have been used for validation. In this work we deepen into these models, with a formal approach to exploration in terms of simulation of such models and we develop a command line and a graphical tool in order to facilitate working with such structures. Furthermore we conducted a case study research in order to validate our approach.

Resumen

Especificaciones mediante pre/postcondiciones son comunes en una variedad de actividades de ingeniería de software, desde requerimientos, pasando por diseño hasta implementación. La naturaleza fragmentada de estas especificaciones pueden dificultar la validación ya que es difícil evaluar si las varias operaciones funcionan adecuadamente juntas. En trabajo previo se han utilizado para validación abstracciones construídas en forma de modelos de comportamientos sintetizados a partir de especificaciones basadas en pre/postcondiciones. En este trabajo ahondamos en estos modelos mediate un acercamiento formal hacia la exploración de los mismo en términos de simulación sobre ellos. Desarrollamos una herramienta de línea de comandos y una visual para facilitar la utilización de dichas estructuras. Para evaluar el enfoque en esta tesis se han llevado a cabo casos de estudios.

Contents

Chapter 1

Introduction

Formal specifications are getting more important each day as software systems evolve and get more complex. They usually disambiguate systems requirements and allow other techniques to be used, such as formal automatic verification and bug finding.

It is usually accepted that formal specification helps build more robust, maintainable software with fewer bugs and defects. Pre/postcondition specifications are being increasingly used and considered good practice in many software engineering activities [17].

In pre/postcondition specifications, a precondition is an assertion that holds before the occurrence of the operation (methods, procedures, use cases, events, etc) and the postcondition is an assertion that holds after the occurrence of the operation, given that the precondition held.

ESC/Java [4], Spec# [1] are programming tools that complement software code development practices attempting to find common run-time errors in programs by static analysis of the program. These tools extend the languages by adding preconditions, postconditions and invariants.

Behaviour models are used in software engineering to specify the behaviour of a software-to-be. These models are commonly used to synthesize fragmented behaviour of the system. They can be extracted from requirements specifications [14], use cases, and scenarios [21]. Generally a graphical representation is used, along with their executable semantics, to validate them. These models are hard to build as they usually have an infinite number of states (or finite but really big) making them difficult to understand and validate [15].

This work relies on a technique to create behaviour models from contract specifications that are an abstraction of all possible implementations that satisfy the contract. This technique is presented in [7]. The models constructed can be used to validate pre/postcondition based specifications through inspection, animation, simulation, and model checking. Authors of that study believe that the criteria chosen for abstraction facilitates validation and debugging. Moreover they think that having the possibility to work and experiment with the model will certainly help track back bugs in an intuitive manner.

Throughout this thesis we will show, by means of a case study, that models constructed in [7] are useful to validate a given contract. Furthermore, we will provide the formal underpinnings and tool support in order to conduct inspection, simulation and model checking of those models. The need of formalization along with the possibility to test and draw conclusions prior to the developing of the software motivates this approach.

Contributions of this thesis are:

1. A formal definition of the notions of contract exploration, enabledness projected contract exploration, abstract exploration and the formalization of the problem of getting unreachable states by means of the properties of the abs_I function, presented in Definition 3.4.
2. An algorithm to transform exploration problems to decidability ones.
3. A semi formal case study showing practical advantages of the proposed technique.
4. Tool support to do concrete exploration and symbolic exploration.

The rest of this thesis is organized as follows. The motivation of our work is introduced in Chapter 2. Chapter 3 explains our starting point, introducing the formal previous background. The formal basis of this thesis is developed

in Chapter 4. Chapter 5 presents the tools developed. The case study conducted in order to analyse our approach is exhibited in Chapter 6. Chapter 7 presents the results obtained in the case study and analyzes them. Finally, conclusions and future work are discussed in Chapter 8.

Chapter 2

Motivation

This work is motivated by [7] which presents a technique to construct models from pre/postcondition specifications.

In this section we illustrate the difficulties of validating pre/postcondition specifications using a small example in order to motivate our approach.

Consider the specification of a circular buffer given in Figure 2.1, which will be used throughout this work.

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 rp and wp 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 its maximum size, in which case the writing pointer is set to 0. Reading requires the buffer to have slots with unread data and updates its reading pointer using the same strategy as *write* uses for wp. Finally, the specification includes an invariant which requires the circular buffer to have more than three slots for storing data and requires both pointers to be different and 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.[1].
4. The `write` operation is always enabled after any (positive) number of `read` operations.[2].
5. Making a `write` and a `read` operation should leave the system in the same state prior executing those operations.
6. We cannot invoke more consecutive `write` operations than the size of the buffer.

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.

[1] Allowed number of *write* operations
[2] Allowed number of *read* operations

CircularBuffer

variable a array of integers
variable wp, rp integer

inv $0 \leq rp < |a| \wedge 0 \leq wp < |a| \wedge |a| > 3 \wedge rp \neq wp$
start $|a| > 3 \ \wedge \ rp = |a| - 1 \ \wedge \ wp = 0$

action `write`(integer n)
  pre $(wp < rp - 1) \ \vee \ (wp = |a| - 1 \ \wedge \ rp > 0)$
    $\vee \ (wp < |a| - 1 \ \wedge \ rp < wp)$
  post $rp' = rp \ \wedge \ (wp < |a| - 1 \ \Rightarrow \ wp' = wp + 1)$
    $\wedge \ (wp = |a| - 1 \ \Rightarrow \ wp' = 0)$
    $\wedge \ (a' = \text{updateArray}(a, wp, n))$

action integer `read()`
  pre $(rp < wp - 1) \ \vee \ (rp = |a| - 1 \ \wedge \ wp > 0)$
    $\vee \ (rp < |a| - 1 \ \wedge \ wp < rp)$
  post $rv = a[rp'] \ \wedge \ wp' = wp \ \wedge \ a' = a$
    $\wedge \ (rp < |a| - 1 \ \Rightarrow \ rp' = rp + 1)$
    $\wedge \ (rp = |a| - 1 \ \Rightarrow \ rp' = 0)$

Figure 2.1. Circular buffer

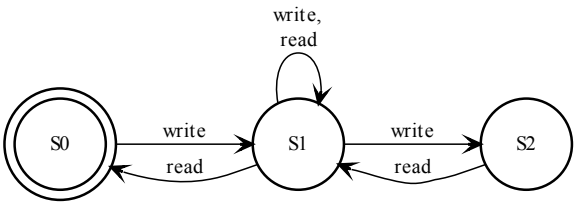

Figure 2.2. Circular buffer finite abstraction

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? How can we ease these checks and get a straight approach to see the results?

It remains to be studied if this automated construction of abstractions [7] that consolidate pre/postcondition specifications into one cohesive behaviour model can complement existing techniques providing support for further analysis and validation of pre/post specifications. Moreover it is interesting to gather evidence if having a way to infer states from traces and aid in this validation process.

Consider, for instance, the behaviour model shown in Figure 2.2 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 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 empty nor full: 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.

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 abstraction level of the model, and the traceability of the abstraction to the specification helps identifying and fixing problems in the latter. Finally it is seen that a simple and fast way to trace a possible sequence of invocations along with the intermediate and resulting states is needed.

In the next chapter we will discuss the formal background in order to build an *Finite State Machine* like the one in Figure 2.2 from a contract like Figure 2.1.

Chapter 3

Background

This chapter presents the formal background and tool support developed in[7] on which this thesis is based.

3.1 Basic Definitions

The following definitions are based on those proposed in[7] and as they will be used in the rest of this thesis, we include them here for the sake of completeness.

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 3.1 (Contract)**.** *A* contract *is a tuple* $C = \langle V,\ inv,\ init,\ A,\ P,\ Q\rangle$ *where:*

- 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. Notice 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 2.1 introduces the contract $C = \langle V,\ \text{inv},\ \text{init},\ A,\ P,\ Q\rangle$ where:

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

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

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

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.

Contract implementations are defined on top of *Data State Machines* (a sort of simplified version of an Action Machine [12]). 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 3.2 (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.*

An implementation of a contract is a DSM that satisfies the contract:

Definition 3.3 (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]*.*

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

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 postcondition of the same action.

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

A possible implementation of contract of Figure 2.1 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:

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

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.

A Finite State Machine is used to provide an abstract representation of a contract, or more precisely, of the implementations allowable by a contract. Simply, an *FSM* is defined as a structure $M = \langle S_a,\ S_{a_0},\ \Sigma,\ \delta \rangle$ where S_a is a finite set of states, $S_{a_0} \subseteq S_a$ is the set of initial states, Σ is a finite alphabet and $\delta : S_a \times \Sigma \rightarrow \wp(S_a)$ is a transition function.

A finite state contract abstraction is an FSM which is able to simulate any possible contract implementation.

Definition 3.4 (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 an FSM $M = \langle S_a,\ S_{a_0},\ \Sigma,\ \delta \rangle$ 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 $abs_I : \mathcal{S} \rightarrow S_a$ such that:*

1. $abs_I(\mathcal{S}_0) \subseteq S_{a_0}$
2. *For every $s \in \mathcal{S}$, and every action label a_i and actual parameter d such that P_{a_i} holds, then $abs_I(\Delta(s, a_i, d)) \subseteq \delta(abs_I(s), a_i)$.*

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.

Definition 3.5 (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.

For instance given the FSCA in Figure 2.2 for a circular buffer of size 4, two enabledness equivalent states, s, t would be:
$s = \{([3,0,0,0], 3, 1)\}$
$t = \{([3,5,0,0], 3, 2)\}$
And they would be equal to the state $S1$ depicted there.

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 3.6 (Enabledness-preserving FSCA)**.** *A Finite State Contract Abstraction $M = \langle S_a,\ S_{v0},\ \Sigma,\ \delta \rangle$ 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 $abs_I : \mathcal{S} \rightarrow S_a$ (a witness abstraction function) such that given a pair of concrete states s, t on $\mathcal{S}$, 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 characterize 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 3.7 (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 3.8 (Enabledness-preserving FSCA by construction)**.** *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 $\mathrm{inv}_{as} \wedge$ init is satisfiable.*
3. *$\Sigma = A$.*
4. *For all $as \in S$ and $a \in \Sigma$, if $a \notin as$ then $\delta(as, a) = \emptyset$, otherwise:*

$$\delta(as, a) \supseteq \left\{ bs \ \middle|\ \mathrm{inv}_{as} \ \wedge Q_a \ \wedge \ \mathrm{inv}'_{bs} \text{ is satisfiable} \right\}$$

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

It is important to note that item 4 of Definition 3.8 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 2.2 no additional transitions due to unfinished satisfiability checks were added [3]

It is straightforward to prove that the FSCA constructed according to Definition 3.8 is an enabledness-preserving abstraction.

Theorem 3.1. *Given a contract C and an interpretation $\langle \mathbb{D}, \mathbb{D}^{op}\rangle$, then M, as built by Definition 3.8, is an enabledness-preserving FSCA of C under the $\langle \mathbb{D}, \mathbb{D}^{op}\rangle$.*

The proof for the theorem can be done simply by 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 $s \equiv_e t \Leftrightarrow abs_I(s) = abs_I(t)$.

Notice that the produced abstractions 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 2.2 but it is not possible to read twice after writing once on any circular buffer implementation independently of its size.

It is important to notice also that some spurious transitions and states are intrinsic to the construction of the FSCA as they have enabledness equivalence, which is comparable to require simulation equivalence for just one step, thus missing information on the history of transitions taken. This could be solved by adding variables and restrictions in the invariant of the contract. In the next chapter we are going to deepen on the subject.

[3] Fortunately, state-of-the-art theorem provers are increasingly able to deal with different "kinds" of formulae in a complete fashion [2, 8] and therefore cases of uncertainty did not arise in any of our case studies.

3.1.1 Tool support

de Caso et al. have developed a tool that parses pre/postconditions contracts and generates an enabledness-preserving FSCA[7].

The tool is commandline based, called *FSCA Generator*, produces several types of graph formats (.dot, xml, etc) and can work with different model checkers.

A contract has the following items:

- **Variable definitions.** The variables that represent structure the model.
- **Model invariant.** A logical formula, having the variables defined earlier.
- **Initial values.** Initial constraints to the variables.
- **Actions**. Each action is composed by:
  - **Action name.** Just a declarative and unique name for that action and different from the constructor
  - **The parameters.** The parameters names and types.
  - **The precondition of the action.** A logical formula that may involve the parameters and the variables.
  - **The postcondition of the action.** A logical formula that may involve the parameters and the variables.

The contract read by the tool must be written in *XML* syntax.

This chapter presented a formal background on which this thesis is based. We continue in the next chapter presenting the formal foundation developed in our work.

Chapter 4

Contract exploration

In the previous chapter we presented a formal model and a technique to automatically construct abstract behaviour models out of pre/postcondition based specifications.

In this section we present the formal underpinnings of our technique to explore and analyze contracts using the enabledness-preserving $FSCA$ presented in the previous chapter.

As mentioned, an enabledness-preserving $FSCA$ is a behaviour model representing all possible implementations that satisfy a given contract.

From now on, when referring to $FSCAs$ we will be referring to enabledness-preserving $FSCAs$.

4.1 Concrete exploration

We will introduce the notion of most general implementation of a contract as all the following definitions are based on the definition of Implementation and we want to formulate the most general ones.

Definition 4.1 (Most general implementation of a contract)**.** *Given a contract implementation* $I = \langle \mathcal{V},\ \mathcal{D},\ \mathcal{A},\ \mathcal{S},\ \mathcal{S}_0,\ \Delta \rangle$ *of* C *under* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ *we say* I *is the most general implementation of* $C = \langle V,\ inv,\ init,\ A,\ P,\ Q \rangle$ *iff*

- $\forall s\ init(s) \Rightarrow s \in \mathcal{S}_0$
- $\forall s, s', a, d\ P_a(s \cup \{p \mapsto d\}) \wedge Q_a(s \cup s' \cup \{p \mapsto d\}) \Rightarrow s' \in \Delta(s, a, d)$

In other words, this is the implementation which represents every possible transition and state within a contract.

Definition 4.2 (Available actions)**.** *We define the set of available actions from a valuation* s *under the implementation* I *as*
$A_{s,I} \stackrel{def}{=} \{a \in \mathcal{A} | \exists d \in \mathcal{D} \wedge \Delta(s, a, d) \neq \emptyset\}$

This is the set of actions which can be executed from a given state. Intuitively, actions which preconditions are met and postconditions lead to a valid state.

Using the example 2.2 of size 4, the state $([0,0,0,0],3,0)$ corresponds to the abstract state s_{a_0}. The set of available actions for that state is `{write}`.

Definition 4.3 (Contract exploration under implementation by construction)**.** *Given a contract implementation* $I = \langle \mathcal{V},\ \mathcal{D},\ \mathcal{A},\ \mathcal{S},\ \mathcal{S}_0,\ \Delta \rangle$ *of a contract* C *under* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$*. We say* E *is a contract exploration under implementation* I *where* E *is constructed as follows:*

1. $s \in \mathcal{S}_0$ *is a contract exploration under* I*.*
2. *Given a contract exploration* $E = s_0, a_0(d_0), s_1, a_1(d_1), ..., a_{k-1}(d_{k-1}), s_k$ *under* I*. The exploration* $E' = E, a_k(d_k), s_{k+1}$ *with* $a_k \in A_{s_k,I}$ *and* $s_{k+1} \in \Delta(s_k, a_k, d_k)$ *is a contract exploration under* I*.*

In other words, a contract exploration under a given implementation can be seen as a path in I starting from any initial state ($s \in \mathcal{S}_0$) and following with an available action from each of the states.

In general, we are interested in analyzing contracts under the most general implementation in order to explore all potential behaviour respecting the contract.

Definition 4.4 (Contract Exploration)**.** *E is a contract exploration of a contract C iff E is a contract exploration under the most general implementation I of C.*

An example of a contract exploration for the circular buffer of Figure 2.2 of size 4 would be the sequence:
$[([0,0,0,0],3,0), write(3), ([3,0,0,0],3,1), write(5), ([3,5,0,0],3,2), read(), ([3,5,0,0],0,2)]$

4.2 Symbolic exploration

This section presents a way of analyzing *FSCAs* in terms of actions and resulting abstract states and not necessarily concrete states (valuations).

Definition 4.5 (FSM Path)**.** *Given an* FSM $M = \langle S_a,\ S_{a_0},\ \Sigma,\ \delta \rangle$.
We define an FSM Path $FP = s_{a_0}, a_1, s_{a_1}, ..., a_i, s_{a_i}, ..., a_n, s_{a_n}$ *where* $s_{a_0} \in S_{a_0}$, *and* $\forall_{1 \leq i \leq n} s_{a_i} \in S_a \wedge a_i \in \Sigma \wedge s_{a_i} \in \delta(a_i, s_{a_{i-1}})$

Definition 4.6 (Enabledness projected contract exploration)**.** *Given a contract C, an implementation I of C, M a $FSCA$ of C and*
$E = s_0, a_0(d_0), s_1, a_1(d_1), ..., a_{k-1}(d_{k-1}), s_k$ *a contract exploration under implementation I*
We define the enabledness projected contract exploration of E, denoted $\alpha(E)$, as the sequence

$$abs_I(s_0), a_0, abs_I(s_1), \ldots, abs_I(s_{k-1}), a_{k-1}, abs_I(s_k)$$

Where abs_I is the total function defined for M

We can see that the enabledness projected contract exploration is a path of abstract states followed by an enabled action for each of those states and we no longer care about the parameters of the actions nor valuations for the system variables (concrete states).

Continuing with the previous example, the enabledness projected contract exploration of the contract exploration depicted in Figure 4.1, would be:

$$[s_{a_0}, write, s_{a_1}, write, s_{a_2}, read, s_{a_1}]$$

Observation 4.1. *From the above definitions it can be observed that:*

- *An enabledness projected contract exploration as in Definition 4.6 is an* FSM Path *of the $FSCA$ of C.*
- *$A_{s,I} = abs_I(s)$ where $A_{s,I}$ is the set of available actions enabled by the valuation s under implementation I, defined in 4.2, and abs_I is the abstraction function of the enabledness-preserving $FSCA$ of contract under implementation I introduced in theorem 3.1*
- *This abs_I is not necessarily surjective neither inyective, as seen in Figures 4.1 and 4.6.*

An example of an enabledness projected contract exploration is depicted in Figure 4.1. We can see an exploration and its projection to the corresponding enabledness-preserving $FSCA$.

Although it is useful to analyze contracts only using their $FSCA$, having the posibility to *explore* the $FSCA$ is a handy complement. Examples of concrete values and real traces can be obtained. These traces would be hard to follow without a symbolic exploration. For instance, we might have to deal with a very large $FSCA$ or one with non-deterministic transitions, which may be hard to explore without appropriate tool support and exploration

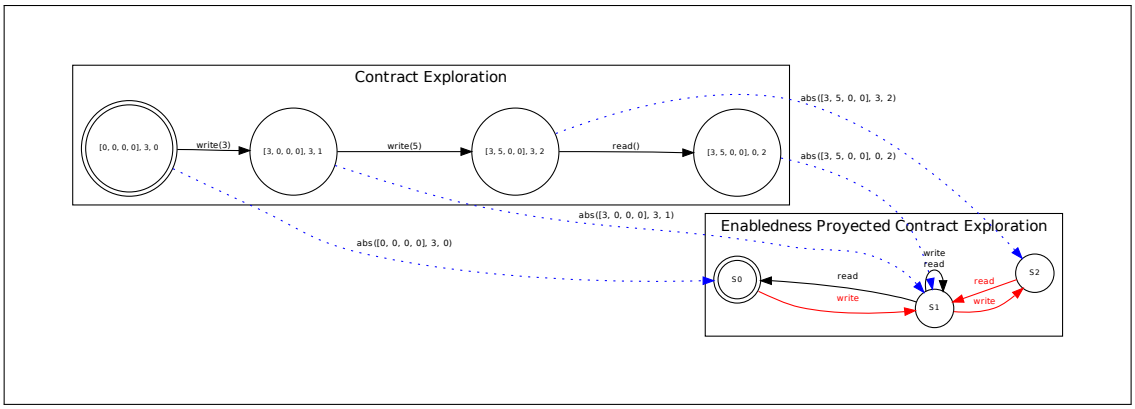

Figure 4.1. Enabledness Projected Contract Exploration Example

capabilities.

Example:
Figures 4.2 and 4.3 show a contract and its corresponding $FSCA$. It would be interesting to see which parameters should be chosen and which values should variables have to be able to take action b3.
As even selecting a parameter value for z the destination state s is unknown, having exploration capabilities is useful.

NonDet
variable x, y integer
inv $y = 25$
start $(x' = 0) \wedge (y' = 25)$
action `a`(integer z)
pre $x = 0$
post $(z = 0 \Rightarrow x' \neq 40) \wedge (z \neq 0 \Rightarrow x' = 40)$
action `b1()`
pre $x = 20$
post *true*
action `b2()`
pre $x = 25$
post *true*
action `b3()`
pre $x = 40$
post *true*

Figure 4.2. Non-deterministic contract example.

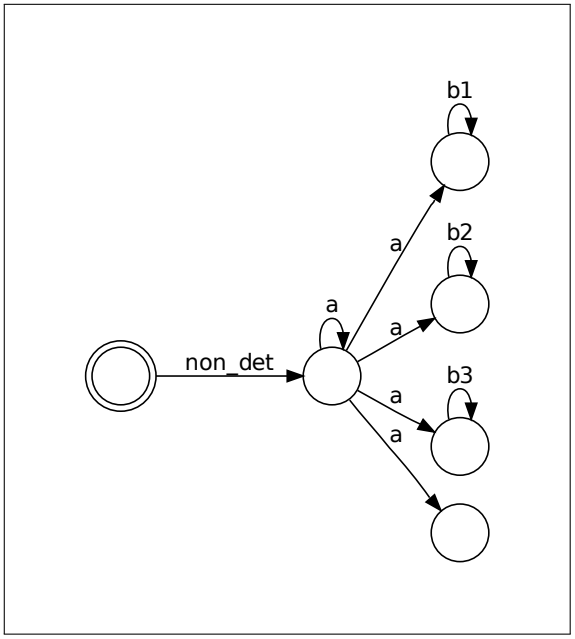

Figure 4.3. $FSCA$ FSM diagram generated by the tool with non-deterministic states

Definition 4.7 (Abstract exploration)**.** *Given an enabledness-preserving $FSCA$ $M = \langle S,\ S_0,\ \Sigma,\ \delta \rangle$ of a contract C we say AE is an abstract exploration of M if AE is a path on the* FSM *M as indicated in Definition 4.5.*

As an example, abstract explorations of the $FSCA$ shown in Figure 4.1 would be:

- $[s_0, write, s_1, write, s_2, read, s_1]$
- $[s_0, write, s_1, write, s_1, write, s_1]$
- $[s_0, write, s_1, read, s_0]$

Observation 4.2. *Given a Contract C, I an implementation of C, M a $FSCA$ of C, E a contract explration under I, then $\alpha(E)$ is an AE. This is illustrated in Figure 4.1.*

Observation 4.3. *There exist AEs that are not projections of any exploration E under any I of C, meaning that there are traces of the $FSCA$ which are not feasible in the concrete model.*
For instance, the following AE from the circular buffer example mentioned in Chapter 2:

$$[s_0, write, s_1, read, s_1]$$

Is not a feasible projection of any exploration under any implementation of the contract shown in Figure 2.1.

4.2.1 Unreachable states

Using an abstraction with a local property as the one used in this work in Definition 3.6, *Enabledness-preserving FSCA*, may generate unreachable states as enabledness equivalence is comparable to require simulation equivalence for just one step.

Figure 4.4 shows an example of a contract that leads to an $FSCA$ with unreachable states.

Unreachable
variable x, y integer
inv *true*
start $(x' = 1) \wedge (y' > 20)$
action `a()`
pre $x \neq 10$
post $x' = 10$
action `b()`
pre $(x \neq 1) \wedge (y < 20)$
post $x' = 2$

Figure 4.4. Contract that generates unreachable states

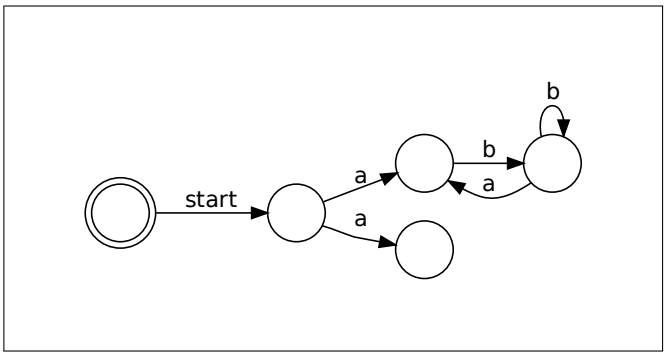

Figure 4.5. $FSCA$ FSM diagram generated by the tool with unreachable states

It can be seen using the exploration tool presented later that there are no possible concrete explorations that lead to take action b, thus getting states and actions unreachable as shown in Figure 4.6
This is because in the *start* action, variable y takes value bigger than 20 and action a, the only action that can then be taken after *start*, does not affect y. So y is never altered by any action that can be taken. Action b has y smaller than 20 as a precondition. Thus action b can never be taken. We can also infer, as y is never altered since *start* action, that $y > 20$ is an invariant thus precondition of action b would violate this invariant.

The fact that abs_I from Observation 4.1 is not necessarily surjective comes from its construction as only one step simulation equivalence is required, as can be seen in Figure 4.6, thus forgetting the history. This unreachability problem is intrinsic to the level of abstraction used. These unreachable states can be eliminated reinforcing the invariant.

In Figure 4.7 we can see the fixed invariant which eliminates the unreachable states.

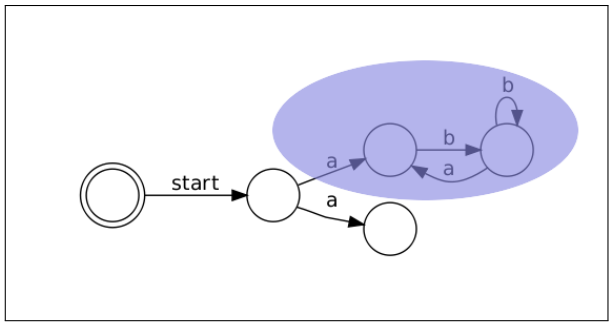

Figure 4.6. $FSCA$ **FSM diagram generated by the tool. Unreachable states and actions are marked**

Unreachable - Eliminated
variable x, y integer
inv $y > 20$
start $(x' = 1) \wedge (y' > 20)$
action `a()`
pre $x \neq 10$
post $x' = 10$
action `b()`
pre $(x \neq 1) \wedge (y < 20)$
post $x' = 2$

Figure 4.7. Fixed contract eliminating unreachable states by restricting the invariant.

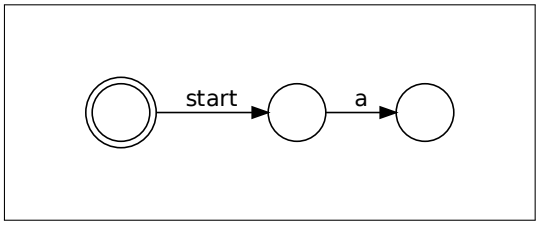

Figure 4.8. $FSCA$ **FSM diagram generated by the tool. Unreachable states were eliminated adding a stronger invariant.**

4.3 Deciding explorations via SMT-solvers

In order to decide if a given exploration (concrete or symbolic one) is consistent with the contract (concrete one) or with the model provided (symbolic one) and obtain resulting states' valuations, SMT-solvers were used [2, 8].

4.3.1 SMT-solvers

An SMT-solver can be used to prove the validity (or, dually, the satisfiability) of first-order formulas in a large number of built-in logical theories and their combination.

SMT-solvers used are automated validity checkers for a many-sorted (i.e., typed) first-order logic with built-in theories, including some support for quantifiers, partial functions, and predicate subtypes. The SMT-solvers used have built-in the theories of:

- equality over free (i.e., uninterpreted) function and predicate symbols,
- real and integer linear arithmetic (with some support for non-linear arithmetic),
- bit vectors,
- arrays,
- tuples,
- records,
- user-defined inductive datatypes.

They check whether a given formula ϕ is valid in the built-in theories under a given set Γ of assumptions. More precisely, they check whether

$$\Gamma \models_T \phi$$

that is, whether ϕ is a logical consequence of the union T of the built-in theories and the set of formulas Γ.

Roughly speaking, when ϕ is universal and all the formulas in Γ are existential (i.e., when ϕ and Γ contain at most universal, respectively existential, quantifiers), they are in fact a decision procedure: it is always guaranteed to return a correct "valid" or "invalid" answer. In all other cases, they are sound but incomplete: they will never say that an invalid formula is valid but they may either never return or give up and return "unknown" in some cases when $\Gamma \models_T \phi$.

4.3.2 Using SMT-solvers in our problem

In order to use an SMT-solver, a transformation from an exploration to a first order formula must be done and sent as a query to the SMT-solver.

In this section we present the transformations used.

4.3.3 Solving concrete explorations

In a concrete exploration, we have a state s_k, an action a_k, a parameter d_k and we want to obtain, if it is a feasible exploration, the resulting state $s_{k+1} \in \Delta(s_k, a_k, d_k)$.

In Figure 4.9 necessary steps and the transformation for each one is shown.

Condition to test	SMT query
Test if s_k satisfies the invariant	$inv(s_k) = true$
Test if $a_k \in A_{s_k,I}$ and if d_k is a legal parameter	$P_{a_k}(s_k \cup \{p_k \mapsto d_k\}) = true$
Get a next concrete state s_{k+1}	$\exists s_{k+1} \mid Q_{a_k}(s_k \cup \{p_k \mapsto d_k \cup s_{k+1}\}) = true$

Figure 4.9. Concrete exploration to satisfiability transformation

In the last step of Figure 4.9, a model is asked to the SMT-solver and the free variable s_{k+1} is obtained.

4.3.4 Solving symbolic explorations

To solve symbolic explorations, also SMT-solvers were used.

Given an FSM Path $FP = s_0, a_1, s_1, ..., a_i, s_i, ..., a_n, s_n$ where $s_0 \in S_0$, and $\forall_{1 \leq i \leq n} s_i \in S \wedge a_i \in \Sigma \wedge s_i \in \delta(a_i, s_{i-1})$ we are interested in analyzing if it is sound and obtain a concrete exploration compliant with it.

In Figure 4.10 we can see conditions and the queries needed to transform from a symbolic exploration problem to a satifiability one.

Condition to test	SMT query
Test if $s_0 \in S_0$	No SMT query is needed. As S_0 is a finite set, it is only needed to check if s_0 is any of those elements.
Given a symbolic exploration, test if it is a valid Enabledness projected contract exploration for the given contract as in Definition 4.6.	$\bigwedge_{a=1..n} \exists d_i \in D.$ $(P_{a_i}(s_{i-1} \cup \{p_i \mapsto d_i\}))$ $\wedge(\exists s_i \mid Q_{a_i}(s_{i-1} \cup \{p_i \mapsto d_i\} \cup s_i))$

Figure 4.10. Symbolic exploration to satisfiability transformation

Having presented our technique's formal model, the next chapter presents the tool we developed.

Chapter 5

Contract Exploration Tool

5.1 Introduction

In the previous chapter we formally defined the notion of enabledness projected contract exploration and abstract exploration. In the present chapter we present the tool developed to automatically evaluate them and augment the capabilities of the contractor tool presented in Chapter 3.

The application was written in Python [11] and its length is about 5000 lines. This tool has four main parts:

- **FSCA generator:** Automatically generates an FSCA from a contract.
- **Concrete explorator:** From a given concrete state (valid valuation) it displays the available actions. Given an available action, a valid parameter [1] and a valuation, one step simulation (lookahead) is done, obtaining the resulting valuation, abstract and concrete state. It returns an error message if the parameters are not sound, the valuation is not a valid concrete state, or the action's precondition is not met with those parameters. This tool can be seen as a simulator for the program described by the contract involved.
- **Symbolic explorator:** From an abstract exploration[2] defined in Chapter 4.7, it can generate a contract exploration as defined in Chapter 4.4 if it is an enabledness projected contract exploration, described in Definition 4.6, or an error if it has unreachable states, transitions or inconsistent restrictions.
- **Graphical User Interface (GUI):** Is a user interface that allows to use the aforementioned tools in an easy and graphical way.

The general architecture of the application is shown in the Figure 5.1

It is worth to mention that the explorator, concrete or symbolic one, does not need to calculate the full *FSCA*, which drastically reduces the time consumed by the tool.

5.1.1 Provided API

A part of the contractor tool was explained in Chapter 3. Each of these applications provides a set of functionality.

- FSCA generator:
  - CreateFSCA. Creates an **FSCA** from a given contract. Its output can be an image (pdf, jpg, .dot file, etc) or an FSCA archive (XML).
- Concrete explorator:

[1] It can be one parameter or an array of parameters. Having multiple parameters does not change the formal foundation as they can be treated as a single one of array type

[2] It can also be a partially-abstract exploration as it can receive a mix of concrete and abstract states.

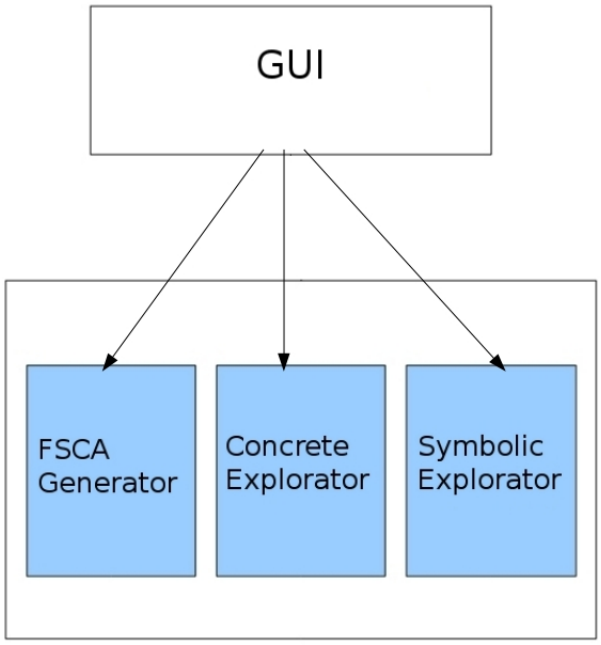

Figure 5.1. Tool general architecture diagram

- GetConstructorsName: Given a contract it gets the constructors names.
- GetStateName: Given a valuation (concrete state), get the state name for that valuation. The states are named after the binary value (converted to decimal) of the activated actions in order of appearance in the contract's XML. For example getting as result the name S1 from the valuation given would mean that the concrete state (valuation) would be projected to a state with only the first action activated[3]. This function is the implementation of the abs function described in Chapter 3.4
- GetAvailableActions: Given a contract and a valuation shows the **available actions** as defined in 4.2 from that state.
- GetNextState: Given a valuation and an action shows next state and a possible valuation for it.

[3]First action in order of appearance in the contract's XML.

  - SimulateAndRuleOut: Given a valuation, an action and a set of valuations shows a next possible state's valuation, different from the optional given ones to rule out. Using this method, many possible outcomes can be evaluated, ruling out ones unwanted ones. This is used when different abstract states are possible after executing the action, meaning is non-deterministic.

- Symbolic explorator:

  - GetConcretePath: Given an **abstract exploration** with optional restrictions on states valuations and actions parameters, get possible valuations for all its variables in each step. Meaning to get one **concrete exploration** from an **Enabledness Projected Contract Exploration**.

The fourth tool is a *GUI* that gives a user friendly access to these other tools and provides a fast way to dig into contracts.

Chapter 6

Case Study

In order to conduct a proper analysis of the work done, possible improvements and future work lines, a case study based research was done. To do so, the guidelines described by Runeson and Host [20] where followed as this approach has been successfully used in many software engineering experiences.

The reason to choose the case study approach is to evaluate empirically the tool and the abstraction in a realistic environment and do it with proven and adequate techniques suitable to research in software engineering.

We understand that empirical studies are critical when evaluating a new model, tool and/or technique. The research has three main goals:

- Firstly, to obtain unbiased information on the usefulness of the abstraction and tools presented in the previous chapters.
- Secondly, to get information on how the abstraction presented is used by outsiders of the research team.
- Thirdly, to provide the research team with useful information on how to improve the work done.

Conducting research on real world issues implies a trade-off between level of control and degree of realism. On the one hand, the realistic situation is often complex and non-deterministic, a fact which hinders the understanding of what is happening, especially for studies with explanatory purposes. On the other hand, increasing the control reduces the degree of realism, sometimes leading to the real influential factors being set outside the scope of the study.

Case studies are by definition conducted in real world settings, and thus have a high degree of realism, mostly at the expense of the level of control.

6.1 Introduction to the methodology

A case study is a suitable research methodology for software engineering research since it studies contemporary phenomena in its natural context.

In this section the methodology and how it is applied in our research is explained.

6.1.1 Purpose on using a Case Study Approach

In[3], the authors distinguish between four types of purposes.

Exploratory: finding out what is happening, seeking new insights and generating ideas and hypotheses for new research.

Descriptive: portraying a situation or phenomenon.

Explanatory: seeking an explanation of a situation or a problem, mostly but not necessarily in the form of a causal relationship.

Improving: trying to improve a certain aspect of the studied phenomenon.

In our research, the purpose of the case study approach is to explore and improve on both the abstraction model and the tool. More particularly, to explore how a researcher would benefit (if at all) from their usage. A secondary goal is to obtain feedback on usage in order to improve them.

6.1.2 Data Collection

Gathered data is going to be used mostly in a qualitative way. As the amount of data to gather is small in order to take a statistic approach, triangulation is used to increase the study's validity.
According to[18] triangulation can be applied in order to secure the effectiveness of data collection.
Four types of triangulation may be applied.

Data source triangulation: using more than one data source or collecting the same data at different occasions.

Observer triangulation: using more than one observer in the study.

Methodological triangulation: combining different types of data collection methods.

Theory triangulation: using alternative theories or viewpoints.

Triangulation types used in the study are:

- Data source triangulation. Data is going to be gathered by having two subjects solve three different problems each. (The same ones each subject).
- Methodological triangulation. Questionnaires (both open and closed) and interviews are used in order to have different types of data collection methods

6.2 Case Study Research Process

In order to conduct a case study, five major processes are to be walked through:

1. Case study design: goals are defined and the case study is planned.
2. Preparation for data collection: procedures and protocols for data collection are defined.
3. Collecting evidence: execution with data collection on the studied case.
4. Analysis of collected data
5. Reporting

6.3 Case Study Design

Although a case study research is of flexible type, planning is necessary. Furthermore, good planning is crucial. According to[3] a plan for a case study should contain at least the following:

- Objective: what to achieve?
  The goal of this study is to explore if the generated model and tools provided are useful to interpret contracts, understand them and find bugs both in the contract vs. specification and in the specification *per se*. For example to find possible contradictions.
  As a second and minor goal the study aims at looking for ways to improve the tool and the model generated by means of the users' experience.

- The case: what is studied?

  In our research multiple units are studied within a case. Particularly, three contracts are going to be studied. All of them are derived from real world specifications. Namely,

  ATM: A simple ATM machine derived from [21].

  MS-NSS: The client role of the Microsoft MS-NSS protocol which is conceived for the negotiation of credentials between a client and a server over a TCP stream[5].

  MS-WINSRA: Windows Internet Naming Service (WINS) Replication and Auto-discovery Protocol[6].

  These units are explained in Appendix A, which is part of the documentation given to the participants of the research.

- Theory: frame of reference

  The frame of reference for this research is the abstraction described in Chapter 3. This abstraction preserves enabledness of sets of operations, resulting in a finite model that is intuitive to validate.

- Research questions: what to know?

  - Does the enabledness abstraction level help to improve the understanding of the contract?
  - Do the exploration tools improve the understanding of the contract and model under study?
  - Do these tools increase the bug finding capabilities of the created model?
  - Do these tools reduce the time required to understand a contract?

- Methods: how to collect data? In [13] three categories of methods: direct (e.g. interviews), indirect (e.g. tool instrumentation) and independent (e.g. documentation analysis) are defined.

  First degree: Direct methods. The researcher is in direct contact with the subjects and collects data in real time. For example, through interviews, focus groups, etc.

  Second degree: Indirect methods. The researcher directly collects raw data without actually interacting with the subjects during the data collection. For example, through automatically monitored software tools and video recording.

  Third degree: Independent analysis of work artifacts. Already available and sometimes compiled data is used. This is for example the case when documents such as requirements specifications and failure reports from an organization are analyzed or when data from organizational databases such as time accounting is analyzed.

  Direct methods (interviews and questionnaires) were selected to gather data. The tool provided could record statistics on how it iss used, but it is not clear how this could be compared with manually studying the contract. Documentation analysis was also used, as every manually written material the participant used to solve the problems was analyzed.

  In case studies, the case and the units of analysis should be selected intentionally. The purpose is to study a particular case, for example one expected to be "typical", "critical", etc.

  For the present research, the units of analysis are selected intentionally in order to explore three levels of contracts (in terms of their size). Participants are also selected to have a similar degree of knowledge and none of them is going to be an expert on the methods or similar models used to evaluate contracts.

  Data collection methods specified are:

  Interviews: The interview will start with open questions and finish with closed ones for the last and bigger example. See Appendix A for the questionnaire.

Observations: Observations can be conducted in order to investigate how a certain task is conducted by software engineers.
The participants were observed every 5 minutes, and for all the case study, in order to see if any relevant data could be gathered by their behaviour.

Archival data: Archival data was gathered by getting all the material the participants generated during their study of the problems.

Metrics: In [10] the authors state that software measurement is the process of representing software entities like processes, products and resources in quantitative numbers.
The definition of what kind of data to collect should be based on a goal-oriented measurement technique. By our goal we infer as relevant metrics two variables; time used in solving the different problems, number of irregularities and bugs found in the contracts.

- Selection strategy - where to seek data? There are several sources of information that can be used in a case study. It is important to use different data sources in order to limit the effects of one interpretation of a single data source. As mentioned earlier, data will be gathered from the questionnaires and observations. Triangulation is going to be used, as mentioned before, in order to have different data sources (participants).

6.4 Ethical Considerations

In [20] the authors mention some ethical considerations regarding measures and documents to sign in order to prevent, mostly legal, problems. Most of them regard confidential information about the organization the case is being held on, the researchers and participants.
The documents required are:

- Informed consent
- Review board approval
- Confidentiality
- Handling of sensitive results
- Inducements
- Feedback

Our study does not need these, as being a degree thesis it is not so sensible to this matters, so we have not consider them in our research.

6.5 Analysis of collected data

Data analysis is conducted differently for quantitative and qualitative data. For quantitative data, the analysis typically includes descriptive statistics, correlation analysis, development of predictive models, and hypothesis testing.

6.5.1 Quantitative Data Analysis

Collected data was:

- Number of questions each participant made about each topic (contracts, FSM viewer and explorator).
- Time required by each participant to solve each problem (if solved).
- Number of problems/issues found on each problem.

6.5.2 Qualitative Data Analysis

In [19] the author states that the basic goal of the analysis is to derive conclusions from the data, keeping a clear chain of evidence. This means that a reader should be able to follow the derivation of results and conclusions from the collected data.

The goal of our qualitative analysis is to find the axes of utility (if any) of the tool and model. These axes would be the kind of issues found (hard or easy to find), ease of use of the tool and type of issue found. The latter would be if there is any type of issue easier to find with the tool than other, for example a precondition that is too weak, etc. Some qualitative information were the questions asked by the participants, the drawings and schemas made by them and the open questions in the questionnaire.

6.5.3 Level of formalism

A structured approach is important in qualitative analysis. All decisions taken by the researcher must be recorded, all versions of instrumentation must be kept, links between data, codes, and memos must be explicitly recorded in documentation, etc. However, the analysis can be conducted at different levels of formalism. In [3] the following approaches are mentioned:

Immersion approaches: These are the least structured approaches, with very low level of structure, more reliant on intuition and interpretative skills of the researcher. These approaches may be hard to combine with requirements on keeping and communicating a chain of evidence.

Editing approaches: These approaches include few *a priori* codes, i.e. codes are defined based on findings of the researcher during the analysis.

Template approaches: These approaches are more formal and include more *a priori* codes based on research questions.

Quasi-statistical approaches: These approaches are much more formalized and include, for example, calculation of frequencies of words and phrases.

According to [20], in their experience, editing approaches and template approaches are most suitable in software engineering case studies. In our study, template approach was used as the main data of the research was gathered through a questionnaire. Also immersion approaches was used, as some qualitative data gathered was based on unstructured observation by the researcher.

6.5.4 Validity

In [20] authors say that the validity of a study denotes the trustworthiness of the results, that is to what extent the results are true and not biased by the researchers subjective point of view.

The scheme proposed distinguishes between four aspects of the validity, which can be summarized as follows:

Construct validity: This aspect of validity reflects to what extent the operational measures that are studied really represent what the researchers have in mind and what is investigated according to the research questions. If, for example, the constructs discussed in the interview questions are not interpreted in the same way by the researcher and the interviewed persons, there is a threat to the construct validity.

In the present study this item is not really a threat as the researcher and the interviewer are the same person. There was a threat that the questions asked would not be properly interpreted by the participants. To mitigate this threat, the questionnaire was reviewed by researchers and students (with different levels of knowledge) prior to the analysis by the participants.

Internal validity: This aspect of validity is of concern when causal relations are examined. When the researcher is investigating whether one factor affects an investigated factor there is a risk that the investigated factor is also affected by a third factor. If the researcher is not aware of the third factor and/or does not know to what extent it affects the investigated factor, there is a threat to the internal validity.

To mitigate this threat, we selected three problems to be solved by the participants and making the research with at least 2 participants for each problem.

External validity: This aspect of validity is concerned with to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the investigated case. During analysis of external validity, the researcher tries to analyze to what extent the findings are of relevance for other cases. There is no population from which a statistically representative sample is to be drawn. However, for case studies, the intention is to enable analytical generalization where the results are extended to cases which have common characteristics and hence for which the findings are relevant, i.e. defining a theory.

To mitigate this threat, two participant where selected and each one tried to solve three different problems. This problems where selected in order to test different contract sizes.

Reliability: This aspect is concerned with to what extent the data and the analysis are dependent on the specific researchers. Hypothetically, if another researcher later on conducted the same study, the result should be the same. Threats to this aspect of validity are, for example, if it is not clear how to code collected data or if questionnaires or interview questions are unclear or ambiguous.

This point is really a threat in our study. To mitigate it, results, questionnaires and conclusions were reviewed by at least an experimented researcher in the field.

6.6 Case Study Protocol

The case study protocol is a container for the design decisions on the case study as well as field procedures for its carrying through. The protocol is a continuously changed document that is updated when the plans for the case study are changed.

In [16] an outline of a case study protocol is proposed, which is summarized in Figure 6.1. The protocol is quite detailed to support a well structured research approach.

Section	Content
Preamble	Contains information about the purpose of the protocol, guidelines for data and document storage, publication.
General	Provides a brief overview of the research project and the case research method.
Procedures	Detailed description of the procedures for conducting each case, including down-to-earth details on contacts and timing.
Research instrument(s)	Interview guides, questionnaires etc. to be used to ensure consistent data collection.
Data analysis guidelines	Detailed description of data analysis procedures, including data schemas, priori codes, etc.
Appendix A	Template letter to invite participants.

Figure 6.1. Case Study Protocol

Although mentioned here seeking completeness, it is not necessary in this study as this document already synthesizes the aforementioned one.

This chapter presented the basis for the case study conducted. In the next chapter an analysis of the results of this case study is presented.

Chapter 7

Analysis

In this chapter an analysis of the results obtained in the case study is presented. Those results are described in the appendixes B, C and obtained by talking to the subjects.

Tests were made as described in Appendix A.

Literal answers by the subjects are displayed in Appendix C.

7.1 Comparison of results

In this section the expected results are compared with those obtained by each participant. These results are also analyzed.

Each particular problem is studied. In the next section more general observations, resulting from observing the participants and interacting with them are discussed.

7.1.1 ATM

The *ATM* example is a simple case study aimed at getting a small and manageable problem. As mentioned in Appendix B some errors were introduced in order to evaluate how the participants work and how many of those errors are found by them.

The first subject, who used the FSCA Generator tool, saw a bug in the contract. He found that in a state reached by *insertCard* and then *enterPassword*, the *withdraw money* action could not be taken. However, we can easily see with the symbolic explorator that this path cannot be taken. After noting that, he mentioned that it was clear that the invariant should be strengthen. The second participant who could not use the FSCA Generator could not see this. To aid himself in attacking the problem, he drew an FSM. It worth noting that this FSM is very similar to the FSCA generated by the tool.

The first subject used the FSCA Generator tool to quickly find bugs and iterate correcting them and trying to find new ones with the new FSCA generated.

The second participant could not even see that there were some mistakes in the invariant (seeing unwanted transitions as mentioned before). He could only find the most superficial bugs or strange behaviours, as having to put the password before the inserting the card, etc.

He also made some mistakes constructing the FSM mentioned earlier, for instance not allowing to take card after inserting it without inserting the password (which is allowed in the contract). These kind of mistakes are surely avoided if assisted by a tool as the *FSM* is constructed by it.

7.1.2 MS-NSS

The *MS-NSS* example is a moderately difficult problem, hard to handle with the contract alone, but still manageable.

We can see that the first participant found some problems, but spent almost all of the time finding them. When allowed to use the FSCA Generator he told us that those problems could have been found almost instantly.

However he could not find more bugs with it. In fact problems found are not bugs, but only features that do not seem to be so intuitive, but are correct.

The second participant, who could use the GUI tool, found some more profound bugs (and real ones) as a weak invariant (a state not reachable) and other ones explained in the next chapter.

Moreover, we saw that the second participant, who had a GUI tool to navigate the FSCA was more motivated to find bugs, change the contract, restart the tool, etc.

7.1.3 MS-WINSRA

The *MS-WINSRA* is a highly complex example with many states and transitions.

Although the second participant tried for over half an hour, he could not even answer the first question using the FSCA Generator tool alone (and the contract).

This shows how difficult it is to follow a complex contract and how focused the operator has to be.

On the other hand, the other participant who could use the GUI tool (symbolic and concrete explorator) could answer both questions and the first one very quickly.

7.2 Results by the researcher

First of all it is interesting to see the drawing (an FSM) made by participant 1 in Figure C.1 and the one made by participant 2 in Figure C.2. Those graphs are very similar to the FSCAs made by the tool and presented in Figures A.2 and A.3 respectively. This is very interesting because it is an indicator that people would manually make a similar graph to the one generated automatically by the tool.

It was seen that both tools (contractor and explorator) were rapidly embraced by the participants. Both subjects found them easy to use. However, the explorator tool was more difficult to use at a maximum potential and they noted that after using it for a while they could see how they could have used it in previous exercises in order to get faster and better results.

By watching them work it could be seen that the explorator tool was heavily used as a visualization tool and to track pre/postconditions faster than looking into the contract. In MS-NSS the tool was mainly used to make symbolic explorations, but only after a while. To solve MS-WINSRA problems this feature was automatically used. When using the GUI tool they hardly ever read the contract, not even the one in the main screen of the tool. It was used just to see the invariant. This could be interpreted as having all the information needed in the other functionalities of the tool.

Not being able to use the contractor or the explorator tool (using only the contract of the given problem) discouraged the participants. They found somewhat difficult to even start attacking the problems. For both (the MS-NSS and the MS-WINSRA problems) even having the contractor tool discouraged them as they could not follow so many transitions. This point problem is handled by the GUI's tool exploration feature of highlighting the transition and resulting state when hovered.

It could be seen that having the GUI tool made the participant interact highly with the problem and gain productivity.

Asking them for their opinion regarding the tools they remarked:

- They were really pleased by the FSMs produced by the contractor tool (The FSCA abstraction [1]). They found them compact and intuitive.
- They were somewhat confused about "not taking into consideration the history". This produced some unreachable states as mentioned in Section 4.2.1 They did not suspect that it could be solved restricting the invariant.
- They found the GUI tool as a "final product" and a finished one contrary to "other" tools found in the academia which they said are always unfinished and buggy.
- They would have liked a copy and paste feature for the "Exploration history" and "Current valuation" boxes in the GUI tool.

[1] Redactor's note

- They would like both of the tools (The GUI and the FSCA generator tool) to startup faster.

Chapter 8

Discussion and Future Work

8.1 Final Discussion

The work carried out in this thesis shows that the **enabledness-preserving contract abstraction** behavioural model is expressive and intuitive enough to help specialists cope with the difficulty of validating pre/postcondition based contracts. Moreover, we have found that models built automatically by this tool are similar to models generated manually by participants when trying to understand the contracts.

We have also found that, for medium to large sized problems, the *finite state contract abstraction* can be very hard to interpret and analyze. Our research found that having tools to simulate and explore the *FSCA* really helps.

Our case study shows that bugs and mistakes can be found faster using the developed tools. Furthermore, big problems could be attacked when they were unfathomable without these tools.

During this work was found that the *FSCAs* created with the contractor tool had unreachable states as a direct consequence of the enabledness equivalence property. It is not clear if the existence of those states should be seen as a flaw of the technique or a bug in the contract's invariant.

On the one hand, if unreachable states are found, we may want them not to appear in the model generated (FSCA).
On the other hand, those states can be seen as a hint that there is something missing (a bug) in the invariant. In fact it has been seen that those states can always be eliminated if the invariant is enforced.
This way of eliminating them could be very prejudicial to the model as many characteristics of the system end up in the invariant making the contracts really hard to write and understand. However, it could be seen as a correct way to write the contracts, as having a much more complete invariant of the system tends to be really useful. The symbolic explorator tool can help us find those states and with them the "bugs" in the invariant. To do so, a symbolic path can be created and if no concrete path can be found by the tool, then we might find unreachable states.

To sum up, the contributions of this work are:

- The notion of concrete exploration.
- The notion of abstract exploration.
- Theoretical background to relate those notions to previous FSCA work.
- Deepening on FSCA study.
- Further analysis of unreachable states and generation of a tool and methodology to find them.
- The concrete exploration tool.
- The symbolic exploration tool.
- A GUI to easily use the tools developed.

8.2 Future Work

In this thesis we started presenting a behavioural model (*FSCA*) to do model checking, analyze contracts and find bugs between a contract and a specification. After that we presented a new approach maintaining the previous one but expanding it with exploration capabilities.

Tools to do one step concrete and multi-step symbolic exploration were also developed.

The usefulness of our approach was tested by conducting a semi formal case study research.

During this work we found a rich field of possibilities in order to expand and continue it. Here we present the most interesting ones found.

- Tool improvements:
  - Search and focus for transitions and states. We found that for big FSCAs searching for a particular state or transition turns out to be really useful.
  - Save and restore features in order to continue an exploration in another moment.
  - A way to export the history of an exploration and the present concrete state in order to analyze them or use them in research documents.
- Theoretical improvements:
  - A way to slice the FSCA. Meaning perhaps a way of expanding and collapsing states, making a "hierarchical" FSCA.
  - A way to make reachability tests. For example, to find if there exists a concrete path from one state to a final one (without having to explicitly define a symbolic path). To do this it would be necessary to specify a number of maximum cycles to explore as it is an undecidable problem.

Appendices

Appendix A

Questionnaire and descriptions for participants of the case study

A.1 Description of the systems

A.1.1 ATM

The system described is a small and simple ATM Machine as presented in[21].

The machine displays a main screen and stays requesting for a password. The user can then either enter a password or insert a card. Whenever a card is inserted the user can issue a canceledMessage, meaning he wants to rollback the operation.

At insertion, a card number is entered. This number must be greater than 0.

When the user wants to eject the card he issues an ejectCard action, the system responds with a requestTakeCard message and then a takeCard action can be executed.

The password is requested (requestPassword) at any time as long as it hasn not been inserted previously. Passwords must be greater than 0. When both password and card are inserted, the user can then withdraw money.

The main screen is displayed whenever there is not a card inserted.

```
<contract name="ATM" invariant="((passwd = 0) <=> (NOT passwordGiven)) AND ((card > 0)
    <=> theCardIn)">
  <variable name="theCardIn" type="BOOLEAN" />
  <variable name="carHalfway" type="BOOLEAN" />
  <variable name="passwordGiven" type="BOOLEAN" />
  <variable name="card" type="INT" />
  <variable name="passwd" type="INT" />

  <constructor name="ATM" pre="TRUE" post="(NOT theCardIn') AND (NOT carHalfway') AND (
      NOT passwordGiven') AND card'=0 AND passwd'=0"/>

  <action name="insertCard" pre="NOT theCardIn AND c > 0" post="theCardIn' AND  card' = c
      ">
    <parameter name="c" type="INT" />
  </action>

  <action name="withdrawMoney" pre="theCardIn AND passwd > 0 AND passwordGiven" post="
      TRUE ">
    <parameter name="m" type="INT" />
  </action>

  <action name="enterPassword" pre="NOT passwordGiven AND q > 0" post="passwordGiven' AND
       passwd' = q" >
    <parameter name="q" type="INT" />
  </action>

  <action name="takeCard" pre="carHalfway" post="(NOT carHalfway') AND (NOT theCardIn')"
      />

  <action name="displayMainScreen" pre="(NOT theCardIn) AND (NOT carHalfway)" post="TRUE"
       />

  <action name="requestPassword" pre="NOT passwordGiven" post="TRUE" />

  <action name="ejectCard" pre="theCardIn" post="(NOT theCardIn') AND (carHalfway') AND
      card'=0 AND passwd'=0 " />

  <action name="requestTakeCard" pre="carHalfway" post="TRUE" />

  <action name="canceledMessage" pre="theCardIn" post="TRUE" />

</contract>
```

Figure A.1. The contract for the ATM example.

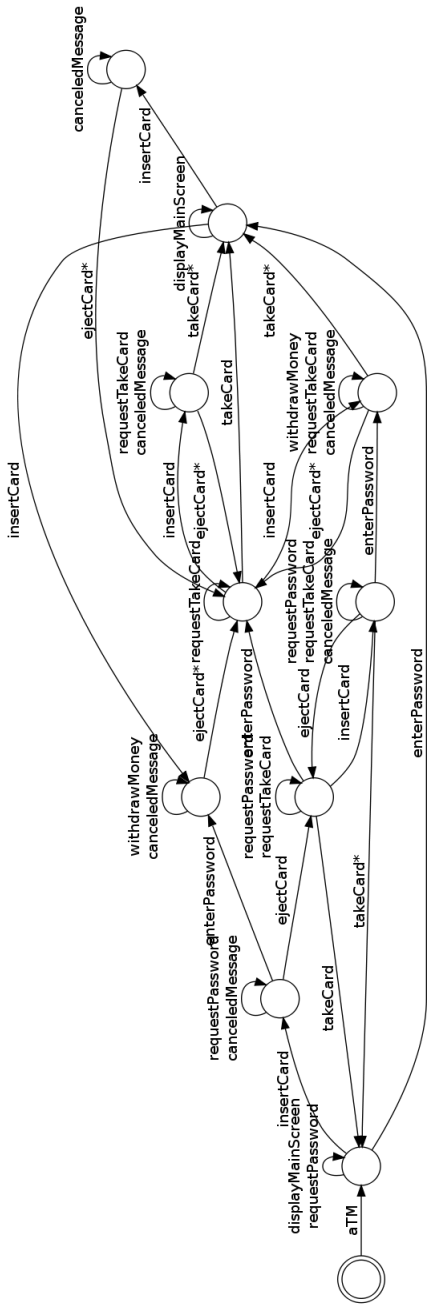

Figure A.2. FSM generated by the FSCA Generator for the ATM example

Solving Objective
Find any bugs in the given contract vs. the specification provided.

A.1.2 MS-NSS

The system is a simple protocol based on Microsoft's NSS .net protocol[5]

The MS-NSS protocol was 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 and

(ii) a data transfer phase in which actual data is transmitted according to the negotiated standards.
In this case study we will only analyze the client role.
The client role in the .NET NegotiateStream protocol initialization is triggered by the application.
The initialization process is as follows:

1. It is the job of the application running in the client role to first establish a TCP connection to the server.
2. Then, before writing any data to the TCP stream, the client triggers the start of the protocol's handshake phase (at any time after the TCP connection is established) by generating and sending a Handshake message. The Handshake message sent MUST be a HandshakeInProgress message. The AuthPayload field of this initial HandshakeInProgress message contains the security token returned from a successful call to the gss_init_sec_context function.
3. If the gss_init_sec_context function returns an error code, then the client MUST create a HandshakeError message.
4. If the gss_init_sec_context function completes successfully, the client SHOULD proceed to the data transfer phase of the protocol.
5. If the client chooses not to proceed to the data transfer phase, then the client MUST create a HandshakeError message.
6. If the HandshakeError message is successfully sent to the server, then the client may retry authentication by repeating the initialization process from step 2.

Handshake Phase
At any point in time, the client is either expecting a Handshake message or a Data Message from the server.
After the initiation of the protocol by the client, it waits to receive a response from the server before continuing the handshake phase of the protocol.
Upon receiving a response the following actions MUST be taken:

- If the number of bytes received from the server is 0: This indicates that the server has initiated the closure of the underlying TCP connection. The client MUST treat the stream as being in an invalid state, and MUST no longer send or attempt to receive further .NET NegotiateStream messages on the TCP connection. The client SHOULD close the TCP connection at this point.
- If the response contains more than 0 bytes: The client MUST check the first byte of the message to see if it matches one of the three known message IDs for .NET NegotiateStream Handshake messages. If the message ID received matches the message ID for:
  - A HandshakeInProgress message: Upon receipt of a message of this type, the client MUST take the token from the AuthPayload field of the message and create a reply Handshake message by passing the token to the gss_init_sec_context function.
    If this function generates an output token, it MUST be sent back to the server in the client's reply. The type of Handshake message that the client replies with depends on the result returned from the gss_init_sec_context function call.
    If the return code represents that the security context is complete, then the client MUST reply with a HandshakeDone message.

  If the return code represents that the security context negotiation is not yet complete, the client MUST reply with a HandshakeInProgress message.
  Finally, if the return code represents that an error has occurred, the client MUST respond with a HandshakeError message.
- A HandshakeDone message: The client MUST proceed to the data transfer phase.

- A HandshakeError message: This message type can be received at any time during the handshake phase and therefore cannot, by definition, be received out of order. The client SHOULD then inspect the error code in the payload. If the error code is equal to 0x8009030C (LogonDenied) or 0x6FE (TrustFailure), then the client may restart the handshake phase or the client may close the TCP connection. For any other error code, the server SHOULD close the connection. The client MUST NOT repeatedly retry the handshake with the same failing credentials.

- None of the preceding handshake message types: The message type is invalid and the client SHOULD restart the handshake phase or close the connection.

Data Transfer Phase

After the server receives a HandshakeDone message from the client and the handshake phase of the .NET NegotiateStream Protocol completes, the protocol MUST transition to the data transfer phase.
At this stage the server or the client can terminate the connection.
In this phase, the server may send a DataMessage to the client at any time, and MUST expect to receive a DataMessage from the client at any time. Upon receiving a message from the client, the server SHOULD inspect the security context negotiated during the authentication phase:

- If the handshake phase resulted in a security context that does not support message integrity or confidentiality, then the data received by the server is not framed, and all bytes read SHOULD be treat as application-level data and passed unmodified to the application.

- If the handshake phase resulted in a security context that supports data confidentiality, integrity or both, then the server SHOULD expect to receive a DataMessage that is framed with the security context negotiated. If the server does not receive a valid DataMessage, it MUST treat the stream as being in an invalid state, and MUST no longer send or attempt to receive further .NET NegotiateStream messages on the TCP connection. The server MUST close the TCP connection at this point.

Here we present the contract's XML:

```
<contract name="MS_NSS_client" invariant="(secContext=-1 OR secContext=0 OR secContext=1)
          AND (0>=requested_protection_level AND requested_protection_level<=3)
          AND (0<=achieved_protection_level AND achieved_protection_level<=3)
          AND (-1 <= handShakeState AND handShakeState <=4)
          AND (NOT tcpConnection => handShakeState <= 0)
          AND (handShakeState = -1 => NOT tcpConnection)" >

  <!-- State of the handshake protocol: 0 = NotStarted, 1 = WaitingResponse,
       2 = Processing, 3 = Proccessed, 4 = Done, -1 = Error -->
  <variable name="handShakeState" type="INT" />

  <!-- Response from gss API: 0 = Complete, 1 = InProcess, -1 = Error -->
  <variable name="secContext" type="INT" />

  <!-- TCP connection state: TRUE = open, FALSE = closed -->
  <variable name="tcpConnection" type="BOOLEAN" />

  <!-- Protection levels (requested and achieved): 0 = None, 1 = Signed, 2 = Encripted,
     3 = SignedEncripted -->
  <variable name="requested_protection_level" type="INT" />
  <variable name="achieved_protection_level" type="INT" />

  <!-- Constructor -->
  <constructor name="establishStream"
    pre="0 <= rpl AND rpl <= 3"
    post="secContext'=1 AND (NOT tcpConnection') AND requested_protection_level'=rpl AND
        achieved_protection_level'=0 AND handShakeState'=0" >
    <parameter name="rpl" type="INT" />
```

```
</constructor>

<!-- Open TCP and initiate protocol -->
<action name="openTCPConnection" pre="handShakeState=0 AND NOT tcpConnection" post="
    tcpConnection' " />

<!-- Close TCP and reset protocol -->
<action name="closeTCPConnection" pre="tcpConnection"
            post="NOT tcpConnection' AND handShakeState'=-1" />
<!-- Call the gss API after receiving a message in progress -->
<action name="gssInitSec"
     pre="tcpConnection AND (handShakeState=2 OR handShakeState=0)"
     post="handShakeState'=3 AND (secContext'=-1 OR secContext'=0 OR secContext'=1)
       AND (secContext'=0 => (0 <= achieved_protection_level' AND
           achieved_protection_level' <= requested_protection_level) )
       AND (secContext'/=0 => (achieved_protection_level' = achieved_protection_level) )
           " />

<!-- Tell the server that we have to continue handshaking -->
<action name="SendHandShakeInProgress"
     pre="tcpConnection AND handShakeState=3 AND secContext=1"
     post="handShakeState'=1" />

<!-- Tell the server that we finished handshaking -->
<action name="SendHandShakeDone"
     pre="tcpConnection AND handShakeState=3 AND secContext=0"
     post="handShakeState'=1" />

<!-- Tell the server that gss API produced an error -->
<action name="SendHandShakeError"
     pre="tcpConnection AND handShakeState=3 AND secContext=-1"
     post="(handShakeState'=0 AND tcpConnection') OR (handShakeState'=-1 AND NOT
         tcpConnection')" />

<!-- The server tells us that the handshake is still to be continued -->
<action name="ReceiveHandShakeInProgress"
     pre="tcpConnection AND handShakeState=1 AND secContext=1"
     post="handShakeState'=2" />

<!-- The server tells us that the handshake phase is over -->
<action name="ReceiveHandShakeDone"
     pre="tcpConnection AND handShakeState=1"
     post="handShakeState'=4" />

<!-- The server tells us that the handshake phase failed -->
<action name="ReceiveHandShakeError"
     pre="tcpConnection AND handShakeState=1"
     post="(handShakeState'=0 AND tcpConnection') OR (handShakeState'=-1 AND NOT
         tcpConnection')" />
<!-- When the handshake is over, we cand send and receive messages -->
<action name="SendData"
     pre="tcpConnection AND handShakeState=4 AND achieved_protection_level=0"
     post="TRUE" />

<action name="ReceiveData"
     pre="tcpConnection AND handShakeState=4 AND achieved_protection_level=0"
     post="TRUE" />
```

```
  <!-- If the achieved protection is adequate, we can exchange cifered messages -->
  <action name="ReceiveValidateData"
      pre="tcpConnection AND handShakeState=4 AND achieved_protection_level > 0"
      post="(handShakeState'=-1 AND NOT tcpConnection') OR (handShakeState'=4 AND
          tcpConnection')" />
  <action name="SendValidateData"
      pre="tcpConnection AND handShakeState=4 AND achieved_protection_level > 0"
      post="(handShakeState'=-1 AND NOT tcpConnection') OR (handShakeState'=4 AND
          tcpConnection')" />
</contract>
```

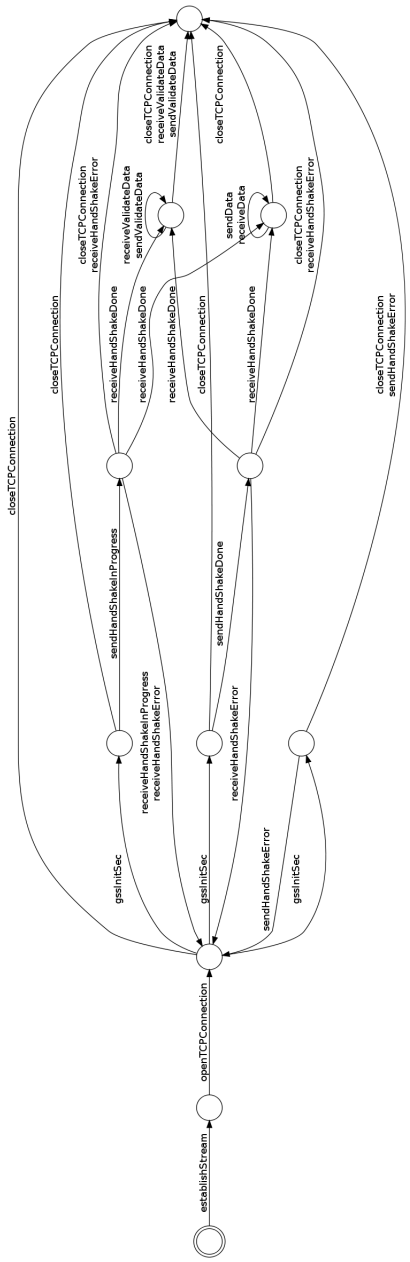

Figure A.3. FSM generated by the FSCA Generator for the MS-NSS example

Solving Objective

- Find any bugs in the given contract vs. the specification provided.
- Find any inconsistencies in the specification provided.
- Find any errors with the contract abstraction.

A.1.3 MS-WINSRA

This system is based on the MS WINS protocol[6] .

Windows Internet Name Service (WINS) is the Microsoft implementation of NetBIOS Name Server (NBNS), a name server for NetBIOS names.

An NBNS server resolves NetBIOS names to the IPv4 addresses that have been registered for that name.

In this example a contract for the protocol is provided.

```
<contract name="MS-WINSRA"
  invariant="2 >= osType AND osType >= 1 AND 2 >= persistent AND persistent >= 1 AND 3 >=
      propagationType AND
  propagationType >= 1 AND 3 >= replicationType AND replicationType >= 1 AND 4 >=
      association AND association >= 1 AND
  17 >= protocolState AND protocolState >= 0 AND ownerVerMergedMap_count >= 0 AND (NOT
      isSetupInitialized => NOT replicationOn)" >
  <variable name="isSetupInitialized" type="BOOLEAN" />
  <variable name="replicationOn" type="BOOLEAN" />
  <variable name="osType" type="INT" />
  <variable name="persistent" type="INT" />
  <variable name="propagationType" type="INT" />
  <variable name="replicationType" type="INT" />
  <variable name="ownerRecordRequested_isNull" type="BOOLEAN" />
  <variable name="association" type="INT" />
  <variable name="protocolState" type="INT" />
  <variable name="ownerVerMergedMap_count" type="INT" />

  <constructor name="Winsra"
    pre="TRUE"
    post="osType' = 2 AND NOT isSetupInitialized' AND 3 >= propagationType' AND
        propagationType' >= 1 AND
     NOT replicationOn' AND association' = 1 AND persistent' = 2 AND replicationType' = 1
         AND protocolState' = 0 AND
     ownerRecordRequested_isNull' AND ownerVerMergedMap_count' = 0" />

  <action name="setupInitialization"
    pre="protocolState = 0 AND NOT isSetupInitialized"
    post="isSetupInitialized' AND 2 >= osType' AND osType' >= 1" />

  <action name="initiateTrafficPull"
    pre="(protocolState = 0 OR protocolState = 8 OR protocolState = 12) AND
        isSetupInitialized AND NOT replicationOn"
    post="((protocolState = 0 => protocolState' = 1) AND (protocolState /= 0 =>
        protocolState' = protocolState)) AND
    replicationType' = 1 AND propagationType' = 1 AND replicationOn' AND (osType = 1 =>
        persistent' = 1) AND
    (osType /= 1 => 2 >= persistent' AND persistent' >= 1)" />

  <action name="initiateTrafficDataVerify"
    pre="(protocolState = 0 OR protocolState = 8 OR protocolState = 12) AND
        isSetupInitialized AND NOT replicationOn"
    post="((protocolState = 0 => protocolState' = 1) AND (protocolState /= 0 =>
        protocolState' = protocolState)) AND
    replicationType' = 2 AND propagationType' = 1 AND replicationOn' AND (osType = 1 =>
        persistent' = 1) AND
    (osType /= 1 => persistent' = 1) AND NOT ownerRecordRequested_isNull'" />

  <action name="initiateTrafficPush"
```

```
    pre="(protocolState = 0 OR protocolState = 8 OR protocolState = 12) AND
        isSetupInitialized AND NOT replicationOn"
    post="((protocolState = 0 => protocolState' = 1) AND (protocolState /= 0 =>
        protocolState' = protocolState)) AND
    replicationType' = 3 AND (propagationType' = 2 OR propagationType' = 3) AND
        replicationOn' AND
    (osType = 1 => persistent' = 1) AND (osType /= 1 => 2 >= persistent' AND persistent'
        >= 1)" />

  <action name="associationStartRequestControlSuccess"
    pre="protocolState = 1 AND replicationOn AND ((association = 1 => persistent = 1) AND
    (association = 2 => replicationType = 3) AND (association = 3 => replicationType /=
        3) AND association /= 4)"
    post="protocolState' = 2" />

  <action name="associationStartRequestControlDiscard"
    pre="protocolState = 1 AND replicationOn AND ((association = 1 => persistent = 1) AND
    (association = 2 => replicationType = 3) AND (association = 3 => replicationType /=
        3) AND association /= 4)"
    post="TRUE" />
  <action name="associationStartRequestControlDisconnect"
    pre="protocolState = 1 AND replicationOn AND ((association = 1 => persistent = 1) AND
    (association = 2 => replicationType = 3) AND (association = 3 => replicationType /=
        3) AND association /= 4)"
    post="protocolState' = 1 AND NOT replicationOn'" /> <!-- PUSE en la pre: "
        protocolState = 1" -->

  <action name="associationStartRequestObserve"
    pre="protocolState = 1 AND replicationOn AND ((association = 1 => persistent = 1) AND
    (association = 2 => replicationType = 3) AND (association = 3 => replicationType /=
        3) AND association /= 4)"
    post="protocolState' = 3" /> <!-- PUSE en la pre: "protocolState = 1" -->

  <!-- 9* -->
  <action name="associationStartResponseControlSuccess"
    pre="protocolState = 3"
    post="protocolState' = 4 AND (replicationType = 3 AND association = 2 => association'
        = 4) AND
    (replicationType = 3 AND association /= 2 => association' = 3) AND (replicationType
        /= 3 AND
    association = 3 => association' = 4) AND (replicationType /= 3 AND association /= 3
        => association' = 2)" />

  <!-- 10* -->
  <action name="associationStartResponseControlDiscard"
    pre="protocolState = 3"
    post="TRUE" />

  <!-- 11* -->
  <action name="associationStartResponseControlDisconnect"
    pre="protocolState = 3"
    post="protocolState' = 1 AND NOT replicationOn'" />

  <!-- 12* -->
  <action name="associationStartResponseObserve"
    pre="protocolState = 2"
    post="protocolState' = 5 AND (replicationType = 3 AND association = 2 => association'
        = 4) AND
```

```
    (replicationType = 3 AND association /= 2 => association' = 3) AND (replicationType
        /= 3 AND
    association = 3 => association' = 4) AND (replicationType /= 3 AND association /= 3
        => association' = 2)" />

  <!-- 13* -->
  <action name="associationStopRequestControlSuccess"
    pre="persistent = 1 AND (protocolState = 6 OR protocolState = 7 OR protocolState = 8)
         AND
    NOT ownerRecordRequested_isNull AND ownerVerMergedMap_count = 0"
    post="protocolState' = 9 AND NOT replicationOn' AND (replicationType = 3 AND
        association = 4 => association' = 2) AND
    (replicationType = 3 AND association /= 4 => association' = 1) AND (replicationType
        /= 3 AND
    association = 4 => association' = 3) AND (replicationType /= 3 AND association /= 4
        => association' = 1)" />

  <!-- 14* -->
  <action name="associationStopRequestControlDisconnect"
    pre="persistent = 1 AND (protocolState = 6 OR protocolState = 7 OR protocolState = 8)
         AND NOT ownerRecordRequested_isNull AND
    ownerVerMergedMap_count = 0"
    post="protocolState' = 1 AND NOT replicationOn' AND (replicationType = 3 AND
        association = 4 => association' = 2) AND
    (replicationType = 3 AND association /= 4 => association' = 1) AND (replicationType
        /= 3 AND
    association = 4 => association' = 3) AND (replicationType /= 3 AND association /= 4
        => association' = 1)" />

  <!-- 15* -->
  <action name="associationStopRequestObserve"
    pre="persistent = 1 AND (protocolState = 10 OR protocolState = 11 OR protocolState =
        12) AND
    NOT ownerRecordRequested_isNull AND ownerVerMergedMap_count = 0"
    post="protocolState' = 13 AND NOT replicationOn' AND (replicationType = 3 AND
        association = 4 => association' = 2) AND
    (replicationType = 3 AND association /= 4 => association' = 1) AND (replicationType
        /= 3 AND
    association = 4 => association' = 3) AND (replicationType /= 3 AND association /= 4
        => association' = 1)" />

  <!-- 16* -->
  <action name="ownerVersionMapRequestControlSuccess"
    pre="replicationType = 1 AND (association = 2 OR association = 4) AND (protocolState
        = 5 OR (protocolState = 8 AND
    persistent = 2))"
    post="protocolState' = 14" />

  <!-- 17* -->
  <action name="ownerVersionMapRequestControlDisconnect"
    pre="replicationType = 1 AND (association = 2 OR association = 4) AND (protocolState
        = 5 OR (protocolState = 8 AND
    persistent = 2))"
    post="protocolState' = 1 AND NOT replicationOn' AND (replicationType = 3 AND
        association = 4 => association' = 2) AND
    (replicationType = 3 AND association /= 4 => association' = 1) AND (replicationType
        /= 3 AND
    association = 4 => association' = 3) AND (replicationType /= 3 AND association /= 4
```

```
      => association' = 1)" />

<!-- 18* -->
<action name="ownerVersionMapRequestObserve"
  pre="replicationType = 1 AND (association = 2 OR association = 4) AND (protocolState
      = 4 OR
  (protocolState = 12 AND persistent = 2))"
  post="protocolState' = 15" />

<!-- 19* -->
<action name="ownerVersionMapResponseControlSuccess"
  pre="protocolState = 15"
  post="protocolState' = 11 AND ownerVerMergedMap_count' >= 0 AND (((replicationOn' &lt
      ;=> replicationOn) AND
  NOT ownerRecordRequested_isNull') OR (NOT replicationOn' AND
  (ownerRecordRequested_isNull' &lt;=> ownerRecordRequested_isNull)))" />

<!-- 20* -->
<action name="ownerVersionMapResponseControlDisconnect"
  pre="protocolState = 15"
  post="ownerVerMergedMap_count' >= 0 AND protocolState' = 1 AND NOT replicationOn' AND
  (replicationType = 3 AND association = 4 => association' = 2) AND
  (replicationType = 3 AND association /= 4 => association' = 1) AND
  (replicationType /= 3 AND association = 4 => association' = 3) AND
  (replicationType /= 3 AND association /= 4 => association' = 1)" />

<!-- 21* -->
<action name="ownerVersionMapResponseObserve"
  pre="protocolState = 14"
  post="protocolState' = 7 AND ownerVerMergedMap_count' >= 0 AND (((replicationOn' &lt
      ;=> replicationOn) AND
  NOT ownerRecordRequested_isNull') OR (NOT replicationOn' AND
  (ownerRecordRequested_isNull' &lt;=> ownerRecordRequested_isNull)))" />

<!-- 22* -->
<action name="ownerVersionMapResponseObserveNoPushMap"
  pre="protocolState = 14"
  post="protocolState' = 7" />

<!-- 23* -->
<action name="updateNotificationControlSuccess"
  pre="(replicationType = 3 AND (association = 3 OR association = 4) AND (protocolState
      = 5 OR
  (protocolState = 12 AND persistent = 2))) AND ((persistent = 1 AND propagationType =
      2) OR
  (persistent = 1 AND propagationType = 3) OR (persistent = 2 AND propagationType = 2)
      OR
  (persistent = 2 AND propagationType = 3))"
  post="protocolState' = 10 AND ownerVerMergedMap_count' >= 0 AND (((replicationOn' &lt
      ;=> replicationOn) AND
  NOT ownerRecordRequested_isNull') OR (NOT replicationOn' AND
  (ownerRecordRequested_isNull' &lt;=> ownerRecordRequested_isNull)))" />

<!-- 24* -->
<action name="updateNotificationControlDiscard"
  pre="replicationType = 3 AND (association = 3 OR association = 4) AND (protocolState
      = 5 OR
  (protocolState = 12 AND persistent = 2))"
```

```
    post="TRUE" />

  <!-- 25* -->
  <action name="updateNotificationControlDisconnect"
    pre="replicationType = 3 AND (association = 3 OR association = 4) AND (protocolState
        = 5 OR (protocolState = 12 AND persistent = 2))"
    post="ownerVerMergedMap_count' >= 0 AND protocolState' = 1 AND NOT replicationOn' AND
         (replicationType = 3 AND
    association = 4 => association' = 2) AND (replicationType = 3 AND association /= 4 =>
         association' = 1) AND
    (replicationType /= 3 AND association = 4 => association' = 3) AND (replicationType
        /= 3 AND
    association /= 4 => association' = 1)" />

  <!-- 26* -->
  <action name="updateNotificationObserve"
    pre="replicationType = 3 AND (association = 3 OR association = 4) AND (protocolState
        = 4 OR (protocolState = 8 AND
    persistent = 2))"
    post="protocolState' = 6 AND ownerVerMergedMap_count' >= ownerVerMergedMap_count AND
    (((replicationOn' &amp;lt;=> replicationOn) AND NOT ownerRecordRequested_isNull') OR (NOT
         replicationOn' AND
    (ownerRecordRequested_isNull' &amp;lt;=> ownerRecordRequested_isNull)))" />

  <!-- 27* -->
  <action name="updateNotificationObserveNoPushMap"
    pre="replicationType = 3 AND (association = 3 OR association = 4) AND (protocolState
        = 4 OR
    (protocolState = 8 AND persistent = 2))"
    post="protocolState' = 6" />

  <!-- 28* -->
  <action name="nameRecordsRequestControlSuccess"
    pre="((protocolState = 8 OR protocolState = 6 OR protocolState = 7) AND
        ownerVerMergedMap_count /= 0) OR
    (protocolState = 5 AND replicationType = 2)"
    post="(replicationType /= 2 => ownerVerMergedMap_count >= ownerVerMergedMap_count'
        AND ownerVerMergedMap_count' >= 0) AND
    (replicationType = 2 => ownerVerMergedMap_count' = ownerVerMergedMap_count) AND
        protocolState' = 16" />

  <!-- 29* -->
  <action name="nameRecordsRequestControlDisconnect"
    pre="((protocolState = 8 OR protocolState = 6 OR protocolState = 7) AND
        ownerVerMergedMap_count /= 0) OR
    (protocolState = 5 AND replicationType = 2)"
    post="protocolState' = 1 AND NOT replicationOn' AND (replicationType = 3 AND
        association = 4 => association' = 2) AND
    (replicationType = 3 AND association /= 4 => association' = 1) AND (replicationType
        /= 3 AND
    association = 4 => association' = 3) AND (replicationType /= 3 AND association /= 4
        => association' = 1)" />

  <!-- 30* -->
  <action name="nameRecordsRequestObserve"
    pre="protocolState /= 1 AND ((ownerVerMergedMap_count /= 0 AND (protocolState = 12 OR
         protocolState = 11 OR
    protocolState = 10)) OR (protocolState = 4 AND replicationType = 2))"
```

```
    post="(replicationType /= 2 => ownerVerMergedMap_count >= ownerVerMergedMap_count'
        AND ownerVerMergedMap_count' >= 0) AND
    (replicationType = 2 => ownerVerMergedMap_count' = ownerVerMergedMap_count) AND
        protocolState' = 17" />

  <!-- 31* -->
  <action name="nameRecordsResponseControlSuccess"
    pre="protocolState = 17"
    post="protocolState' = 12 AND (persistent = 2 => NOT replicationOn') AND
    (persistent /= 2 => (replicationOn' &lt;=> replicationOn)) AND NOT
        ownerRecordRequested_isNull'" />

  <!-- 32* -->
  <action name="nameRecordsResponseControlDisconnect"
    pre="protocolState = 17"
    post="protocolState' = 1 AND NOT replicationOn' AND (replicationType = 3 AND
        association = 4 => association' = 2) AND
    (replicationType = 3 AND association /= 4 => association' = 1) AND (replicationType
        /= 3 AND
    association = 4 => association' = 3) AND (replicationType /= 3 AND association /= 4
        => association' = 1)" />

  <!-- 33* -->
  <action name="nameRecordsResponseObserve"
    pre="protocolState = 16"
    post="protocolState' = 8 AND (persistent = 2 => NOT replicationOn') AND
    (persistent /= 2 => (replicationOn' &lt;=> replicationOn)) AND NOT
        ownerRecordRequested_isNull'" />

</contract>
```

The FSM generated by the FSCA Generator for the MS-WINSRA example is too big to be included in this work. For completeness it is available in the thesis webpage[1]

Solving Objective

- When is the system going to a deadlock state from an initiatTrafficPull transition ? Why ?
- Give a trace that can execute an updateNotificationControlDiscard transition. Which conclusions can you make about why that transition can be executed ?

[1] http://www.tcach.com.ar/thesis

A.2 Distribution of participants

As mentioned in the methodology for the case study in chapter 6 two participants took place in the case study.

Both participants have similar background and have been in touch with abstract models and formal contracts to specify software but not with the behavioural model presented in this work.

In order to evaluate the usefulness of the abstract model presented and the tool developed, each subject tried to solve each problem with a different tool than the other. For the rest of this sections participants are going to be called **Participant 1** and **Participant 2**. For each experiment the participants worked with the contract and the specification provided previously in this section. For some of the problems, they could use the other tools. Which tools could each participant use to solve each of the of the problems is detailed in the following section.

A.2.1 Notes for each problem

Here we dig into how each problem was assigned to each participant.

- *ATM* Time given to analyze this problem was 40 minutes.
  The **Participant 1** was given the FSCA generator tool to solve this problem. The **Participant 2** could only use the contract provided.

- *MS-NSS* Time given to analyze this problem was 60 minutes.
  The **Participant 1** could only use the contract for the first 20 minutes. After that he could use the FSCA generator tool. The **Participant 2** was given the GUI tool.

- *MSS-WINSRA* Time given to analyze this problem was 30 minutes.
  The **Participant 1** could use the GUI tool. The **Participant 2** could use the FSCA Generator tool only.

Appendix B

Bugs and problems expected to be found and possible answers to the questions made

B.1 ATM

- The invariant is too weak as you can have both inserted, password and card, but cannot withdrawMoney.
- Insert card should have cardHalfway' = false in the contract.
- The invariant is too weak as doesn nott have that card>=0 and password >=0.
- The contract is wrong because a cancel message could be issued and then a withdrawMoney. It could be fixed by setting a Boolean variable being cancelled.

B.2 MS-NSS

- It can go to a deadlock state without closing the TCP connection.
- It can reach a deadlock state with send and receive validate data.
- It can be noted that sendValidateData cannot be reached only with one pass from gssInitSec. (From S450 to S6146 in the FSCA). That is a problem of the abstraction made by Contractor. This is explicitly forbidden in the original specification.

B.3 MS-WINSRA

1. When is the system going to a deadlock state from an initiatTrafficPull transition ? Why ?
   It can be seen that state *S0* shown in the FSCA is a deadlock one. We can then see all the transitions entering to it and where they come from. We can even start an exploration and get the values needed to reach it. We see that to follow the action in question from the init state can take us to states S240 and S0. The only difference is whether the variable persistent takes values 1 or 2. If it has 2, then it goes to a deadlock state, which is not sound, so it is a bug.
2. Give a trace that can execute an updateNotificationControlDiscard transition. Which conclusions can you make about why that transition can be executed ? There are many possible. Seeing the preconditions of the transitions we can see which values must have. With an exploration we can navigate the states according the wanted values and pass through actions that sets the properties to the values we need.
   A possible trace:
   winsra(), setupInitialization(), initiateTrafficPush(), associationStartRequestControlSuccess(), association-StartResponseObserve(), updateNotificationControlDiscard().
   We can see its postcondition is True, so it doesn not change anything, which is strange.

Appendix C

Obtained Results

In this chapter annotations made by the subjects are shown. Tests were made as described in Appendix A

C.1 Results by the subjects

In this section the annotations and comments made by each participant for each experiment are presented. These annotations are literal copies of questions asked and remarks made by the subjects.

C.1.1 Subject 1

ATM

1. Is it possible to insert a password without inserting a card? Wouldn not be more reasonable that requestPassword has as precondition that the card was inserted ?
2. I can see in the FSM ($FSCA$'s graph [1]) that there is a state that has a password and the card, but cannot withdrowMoney.
   With enterPassword, as we have in the precondition that password entered is bigger than 0 ($q > 0$) then with the postcondition, we have: $passwordGiven \wedge password > 0$. Similar with insertCard, getting $theCardIn \wedge card > 0$.
   Then we have that withdrawMoney has pre: $theCardIn \wedge password > 0 \wedge passwordGiven$
   So according to the contracts this transition should be activated in that state.
3. I change the contract with the precondition to have cardIn in requestPassword and enterPassword.
   I see I get a much smaller FSM and a nicer one and easier to see bugs in it.
4. As the "lack of memory" of the FSM I see that I can do: insertCard, takeCard, insertCard and then withdrawMoney.
5. insertCard should have $!cardHalfway$. I correct that and I see a much simpler FSM.
6. I see that with more iterations I could get a better FSM and surely find more bugs.

MS-NSS First 20 minutes, only using the contract. (Without FSCA Generator nor symbolic or concrete explorator). Then using only the FSCA Generator tool The subject makes the following drawing.

1. The client can close the connection without sending the handShakeError.

[1] Added by the editor

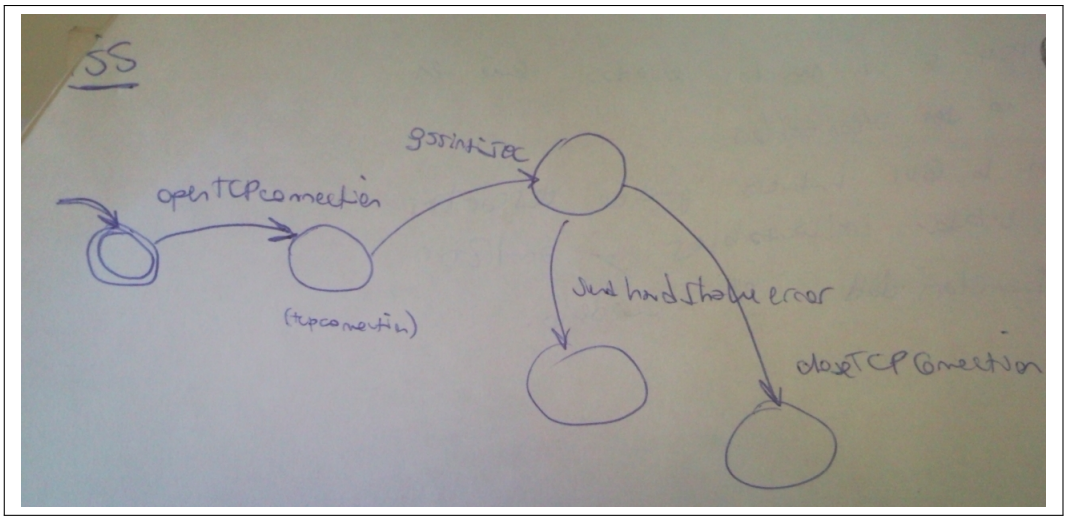

Figure C.1. Draw made by Subject 1 solving the MS-NSS problem

2. It seems that if the handShake ends ok, but the client doesn not want to send data, it is not possible because the precondition of sendHandshakeError.

3. Both bugs found can be seen clearly in the FSM generated by contractor. It would have been easier to find them this way.

4. In the FSM there seems to be a lot of states that are not reachable. Perhaps using the explorator tool I could rule them out and then have a cleaner FSM to analyze and find more bugs.

MS-WINSRA

- When is the system going to a deadlock state from an initiatTrafficPull transition ? Why ?

  The first one is easy. Just executing the explorator.
  I executed the trace:
  winsra(), setupInitialization(), initiateTrafficPull().
  I can see it is quite strange that I can go to a deadlock state just generating an initiateTrafficPull and any other unusual thing. Might be a bug.

- Give a trace that can execute an updateNotificationControlDiscard transition. Which conclusions can you make about why that transition can be executed ?
  winsra(), setupInitialization(), initiateTrafficPush(), associationStartRequestControlSuccess(), association-StartResponseObserve(), updateNotificationControlDiscard().
  Did not have time to analyze it properly, but could take that.

C.1.2 Subject 2

ATM The subject makes the following drawing.

1. Canceled message weird

2. Can put password before card

3. User can take the card before seeing the message requestTakeCard.

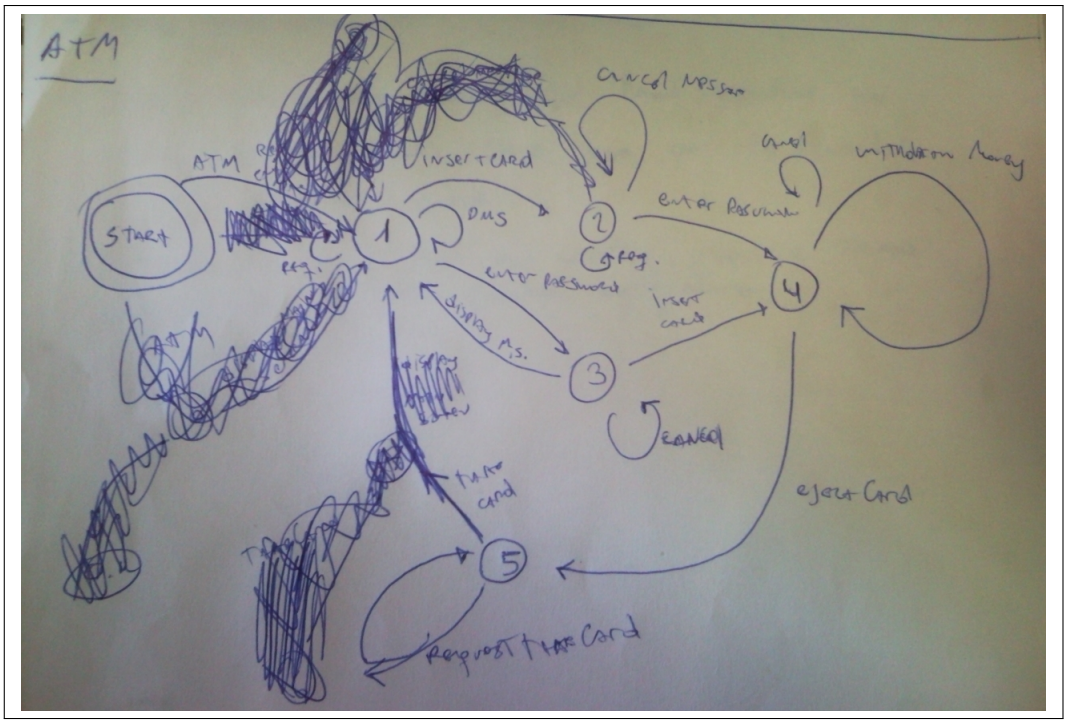

Figure C.2. Draw made by Subject 2 solving the ATM problem

4. passwordGiven does not get cleaned !!. Then ejectCard can invalid the invariant.

MS-NSS

1. Can send a handshakedone without sending a handShakeInProgress
2. It seemed that it could receive a handShakeDone without sending a handShake done, but with the exploration tool I saw that path could not be taken.
3. It is rather a complex case, with more time I think more bugs could be found using the tool.

MS-WINSRA I started to calculate the preconditions and postconditions, but I could not finish the first question in the given time only using the contract and FSCA Generator tool.

Appendix D

Contract Exploration Tool Manual

D.1 Commandline contractor tool

The main tool, contractor, was made as a commandline tool in order to be able to integrate easily, independently and without compilation of any sort into any new application.
This kind of architecture is commonly found in many Linux projects. A very non-extensive list as example is:

- gnomebaker, a GUI cd/dvd recording tool, cdrecord, a commandline cd/dvd recording tool.
- gnomebaker and mpg321, a commandline mpg/mp3 player/reader tool.
- brasero, a GUI cd/dvd recording tool, cdrecord, a commandline cd/dvd recording tool.
- snd-gtk, a GUI for sound manipulation and snd, a commandline sound manipulation tool
- FSCA generator and CVC3[2]

The Graphical User Interface developed executes calls to the contractor tool with different parameters in order to work.

With this architecture, a different GUI tool could be easily be developed, having all the functionality (core functionality) given by contractor, as depicted in the previous list with both brasero and gnomebaker using cdrecord.

The three tools developed (FSCA generator [1], contract_explorator and symbolic_explorator) are part of the same commandline tool, called contractor, but could easily be abstracted into three different ones. They were grouped together just for simplicity for this work.

In fact in a recent re-factor they have already been separated in two. The FSCA generator tool on one side and both the symbolic and concrete explorator on the other.

A full help of the tool can be seen by executing the commandline in Figure D.1

```
contractor --help
```

Figure D.1. Getting help for the contractor tool

The tool needs two main dependencies to work. One is graphviz[9], used to generate the graphs for the *FSCAs* and the other is the SMT-solver and for the moment can be either CVC3[2] or yices[8] but many more can be easily added.

[1] Developed by de Caso et. al[7]

D.1.1 Concrete explorator

The concrete explorator tool allows to "execute" a contract, meaning doing one step simulation of the program described by the contract given.

It is worth to mention that this tool does not require to generate the FSCA, which is very expensive in terms of CPU usage because it has to evaluate all the enabledness states and these are $2^{number\_of\_actions}$.
We have seen that in order to perform a contract exploration it is only required to evaluate the actions' preconditions and postconditions. Thus the tool takes linear time in the quantity of actions, making it quite a fast application.

Passing the tool the "–help" parameter gives a detailed description of usage.

The concrete explorator tool is executed calling contractor with the "-m concrete" parameter.

For instance, using the stack example, we can execute:

```
./contract_explorer.py -m concrete Stack.contract

Available actions: stack
State name: Sinit
```

Figure D.2. Example of executing the concrete explorator

In the previous example, execution of the concrete explorator with no valuation, is presented. As no valuation is entered, it is assumed is in the initial state. As no action is provided, it just shows the state name and the available actions at that state.

Execution result is as follows.

```
./contract_explorer.py -m concrete --action=stack[3] ../examples/Stack.contract

Valuation:
  next = (- 1)
  max = 3
Available actions: push
State name: S2
```

Figure D.3. Example of executing stack action in the concrete explorator

The stack action with 3 as its parameter is executed. The next state name is shown along with its valuation and the available actions from that state. In order to execute that action, the valuation is needed to be specified. This is shown in the next Figure:

```
./contract_explorer.py -m concrete --valuation=-1,3 --action=push[54] ../examples/Stack.
    contract
Valuation:
  next = 0
  max = 3
Available actions: pop push
State name: S3
```

Figure D.4. Example of executing the push action in the concrete explorator

As can be seen, push or pop actions are now available. Its next element marker has been updated. The parameters order in the valuation and actions must be in the same order of appearance as in the contract.

D.1.2 Symbolic explorator

The symbolic explorator tool receives an abstract exploration, defined in 4.7 , augmented with state names and gives as output a possible valuation for each of the states and parameters of actions. It can receive restrictions for the actions, parameter values and states.

A symbolic exploration is performed using the contractor tool with the "-m symbolic" parameter.

```
./contract_explorer.py -m symbolic --path="Sinit->stack->S2->push->S3" ../examples/Stack.
    contract
-- State Sinit

-- Action stack
  size = 3

-- State S2
  next = (- 1)
  max = 3


-- Action push
  elem = 0

-- State S3
  next = 0
  max = 3
```

Figure D.5. Example of executing a symbolic exploration

If the simulation is not possible an error is returned, as depicted in Figure D.6

```
./contract_explorer.py -m symbolic --path="Sinit->stack->S3" ../examples/Stack.contract
Path transition is not possible
```

Figure D.6. Example of executing an unsound symbolic exploration

In order to set restrictions to parameters or states, they are entered between "[]". For instance:

```
./contract_explorer.py -m symbolic --path="Sinit->stack[size=5]->S2" ../examples/Stack.
    contract
-- State Sinit

-- Action stack
  size = 5

-- State S2
  next = (- 1)
  max = 5
```

Figure D.7. Example of executing a symbolic exploration with restrictions

D.2 Graphical User Interface (GUI)

The *GUI* tool is also programmed in Python and as graphic toolkit GTK was used. This decision was made as GTK is a multi-platform toolkit, allowing future migrations of the tool to other operating systems easily.

The tool is executed with the command:

```
./contractor_gui.py --contractor-path=../engine/contractor.py  --contract-explorer-path
    =../engine/contract_explorer.py
```

Figure D.8. Executing the GUI

As mentioned earlier, the *GUI* tool uses the other commandline tools. That is why the contractor path and the contract-explorer path must be passed as parameters.

As with the rest of the tools the parameter "–help" can also be passed in order to get full help. The contract to analyze can also be passed as parameter, or in the main screen the "Open" button can be clicked in order to select a contract to open.

In Figure D.9 the main screen of the *GUI* is shown.

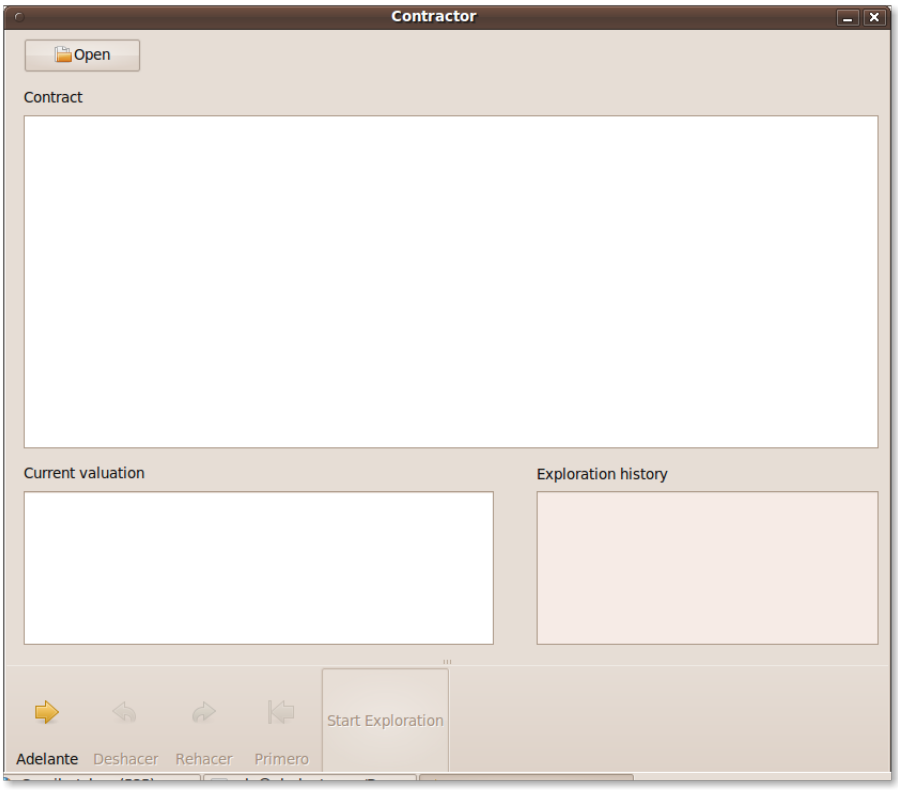

Figure D.9. Gui main screen

The three fields shown in the Figure D.9 mean:

- Contract: In this field we will see the contract opened.
- Current valuation: The valuation of the state being explored. Clicking on each variable its value is displayed,
- Exploration history: The whole history of the simulation made. It is worth noting that in order to be usable it is linearized, as is in a webrowser history. (In fact the navigation history has a tree structure). Clicking on each action its parameters are displayed. Clicking on each state the valuation is shown.

On opening a contract, another window appears, called "Enabledness-preserving contract abstraction", showing the *FSCA* and the Contract field of the main screen is filled with the contract. This can be seen in Figure D.10.

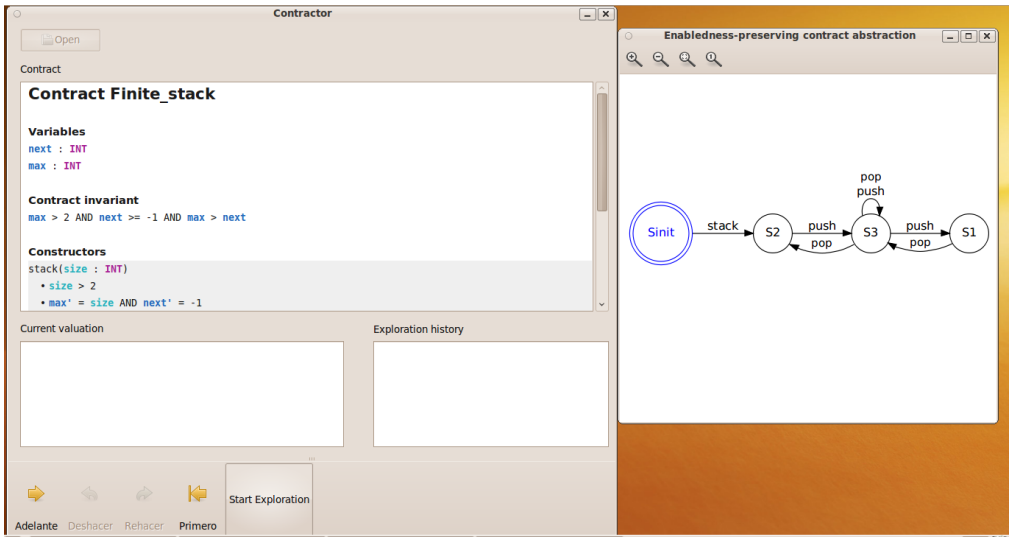

Figure D.10. GUI tool with an opened contract

The state being explored is marked in blue.
In the "Enabledness-preserving contract abstraction" window we can:

- Zoom in/out by using the mouse wheel or using the toolbar on top of that window
- Drag the *FSCA* in order to move it and properly explore it.
- Hovering over a state marks it in red.
- Hovering over a transition marks it in red. If hovered over the beginning of the transition (near the source state) also the destination state gets marked, in order to be able to determine quickly the transition's destination. If hovered over the end of it (near the destination state) the source state gets also marked.
- Left clicking on a transition or state gets it centered.
- Right clicking on a transition displays the transition inspector window showing for each action in that transition its source state, its destination state, the action's precondition, postcondition and modified variables.
- Right clicking on a state displays the state inspector window showing its name, and its enabled actions along with each possible destination.

In the main screen and once a contract is loaded we can:

- Clicking on the "forward" button, perform one step concrete exploration. Then the action selection screen is displayed. If the action has parameters, the parameter selection screen is shown in order to enter a value for each one (separated by commas and in the order of appearance in the contract). After this, the action is performed and the new state is explored.
- Clicking on the "back" button last action is undone. Undo can be done until first action on the history.
- Clicking on the "forward" button we can redo last action.
- Clicking on the "start" button we can undo actions and go back to the first state.
- The "Start Exploration" button functionality is explained later in this section.

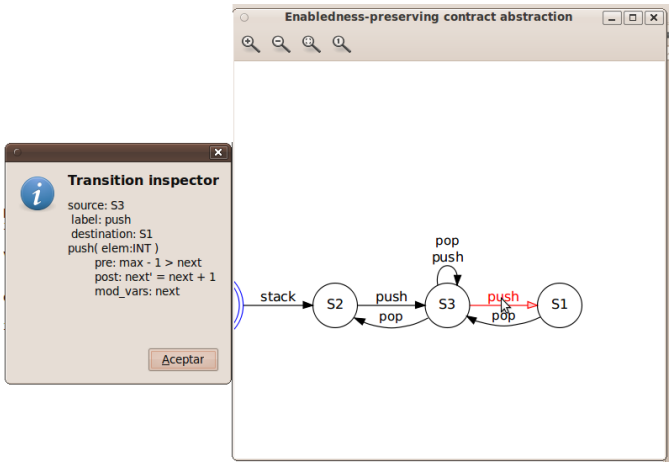

Figure D.11. Transition Inspector window

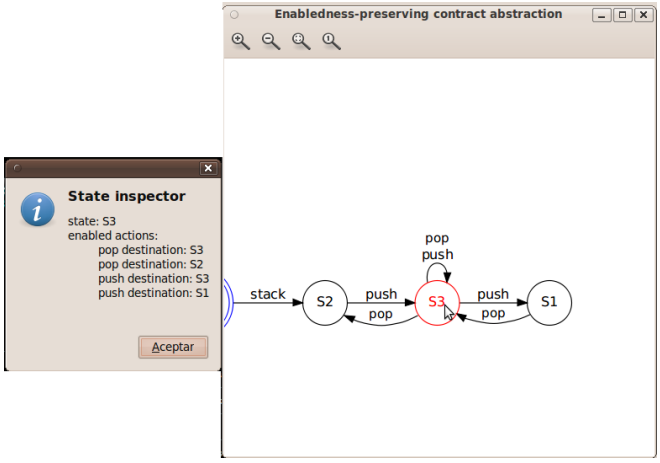

Figure D.12. State Inspector window

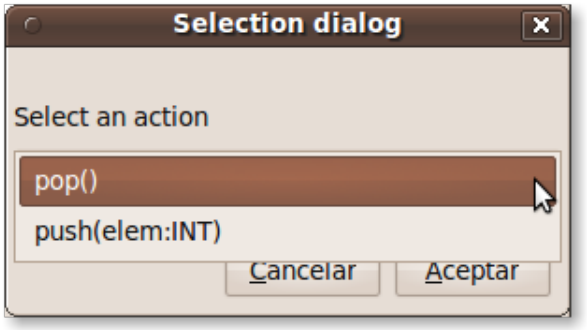

Figure D.13. Action selection window

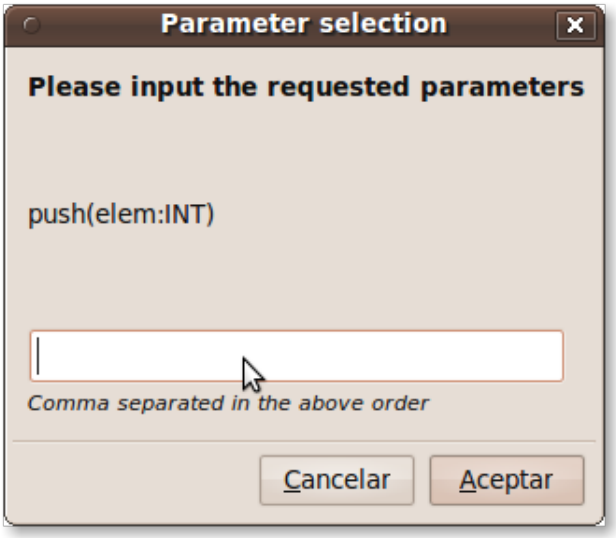

Figure D.14. Parameter selection window

D.2.1 Symbolic exploration

The symbolic exploration button allows to execute easily the symbolic explorator tool.

Once clicked enters the "Exploration mode" which shows the "Path selection" window.

In that window, edition and addition of restrictions to the different actions and states can be done.

In the Exploration mode the Enabledness-preserving contract abstraction window shows the last state marked for symbolic exploration and which actions can be taken from that state. The right mouse button can select the desired action (from the available ones) adding a symbolic exploration step. Selected steps appear in the Selection Path window. After finishing selecting the exploration wanted, it can be executed (sent to the symbolic exploration tool) with the "Send symbolic exploration" button. If the exploration selected is sound, meaning that every pre/post and invariant are valid with the selected values, the Symbolic exploration is made.

If it is not valid, the tool show a message. It will also make a binary search in the exploration path in order to find the last valid step.

Once the symbolic exploration is made, the tool returns to the main screen and shows the last state of the exploration, with the history done and a valid concrete valuation. From there the exploration can be continued.

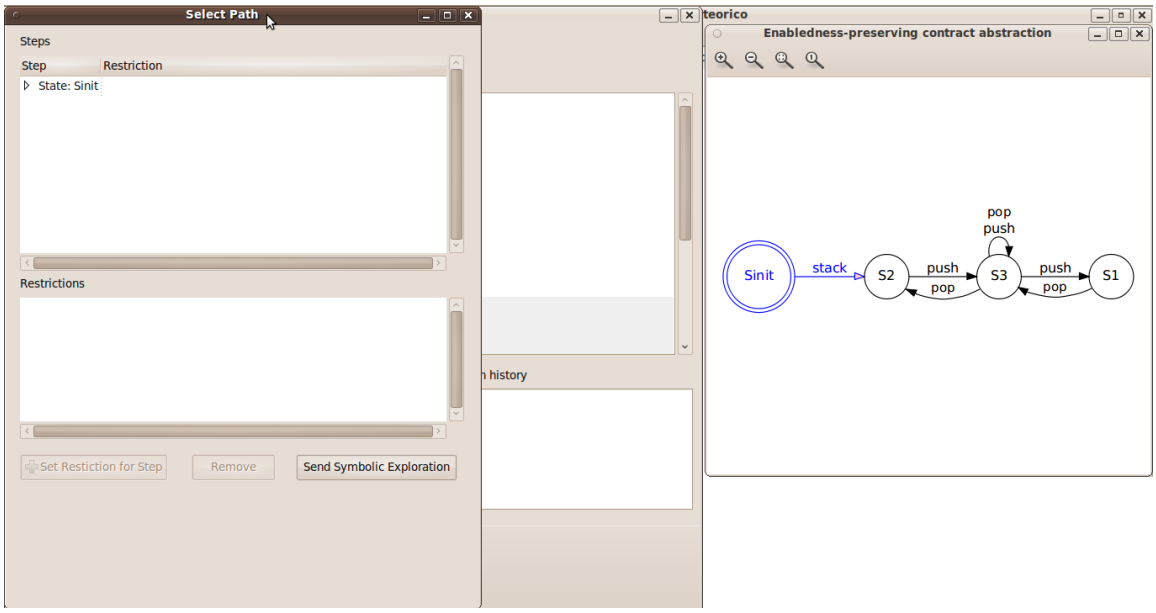

Figure D.15. Path selection window

Bibliography

[1] M. Barnett, K. Leino, and W. Schulte. The SpecSharp programming system: An overview. *Lecture Notes in Computer Science*, pages 49–69, 2005.

[2] C. Barrett and S. Berezin. CVC Lite: A new implementa-tion of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification*, 2004.

[3] R. C. *Real World Research*. Blackwell, 2002.

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

[5] M. Corporation. .NET NegotiateStream Protocol Specification, 2009.

[6] M. Corporation. Windows Internet Naming Service (WINS) Replication and Autodiscovery Protocol Specification, 2009.

[7] G. De Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Validation of Contracs using Enabledness Preserving Finite State Abstractions. *ICSE 09*, 1(1):1, 2009.

[8] L. de Moura and B. Dutertre. Yices 1.0: An efficient SMT solver. *The Satisfiability Modulo Theories Competition (SMT-COMP)*, 2006.

[9] A. B. et al. Graphviz Visualization Tool. http://www.graphviz.org.

[10] P. S. . Fenton N. *Software Metrics. A Rigorous and Practical Approach*. Thomson Computer, 1996.

[11] P. S. Foundation. Python programming language. http://www.python.org.

[12] 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.

[13] S. J. Lethbridge TC, Sim SE. Studying software engineers: data collection techniques for software field studies. *Empir Software Eng*, 10:311–341, 2005.

[14] 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.

[15] J. Magee and J. Kramer. Concurrency: State Models and Java Programs. 1999. *New York: John Wiley & Sons Ltd.[IBMb 02] IBM,"Business Process Execution Language For Web Services", Version*, 1:3–6.

[16] M. H. Pervan G. Designing a case study protocol for application in IS research. *The Ninth Pacific Conference on Information Systems*, pages 1281–1292, 2005.

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

[18] S. RE. *The art of case study research*. Sage, 1995.

[19] Y. RK. *Case study research*. Sage, 2003.

[20] P. Runeson and M. Host. Guidelines for conducting and reporting case study research in software engineering. *Empir Software Eng*, 14:131–164, 2009.

[21] J. Whittle and J. Schumann. “Generating Statechart Designs from Scenarios”. In *ICSE’00*, pages 314–323, 2000.