Università degli Studi di Milano
Dipartimento di Informatica
Scuola di dottorato in Informatica, XXV Ciclo
Dottorato di ricerca in Informatica (INF/01 Informatica)

Tool-Assisted Validation and Verification Techniques for State-Based Formal Methods

Tesi di dottorato di ricerca di
Paolo Arcaini

Relatore
Prof.ssa Elvinia Riccobene

Correlatore
Dott. Angelo Gargantini

Direttore della scuola di dottorato
Prof. Ernesto Damiani

Anno Accademico 2011/12

Acknowledgements

I would like to thank my advisor Elvinia Riccobene for introducing me to scientific research since my master studies; I am grateful for her constant guidance, support and help.

I would like to thank Angelo Gargantini for his continuing and patient support.

Finally, I would like to thank the referees, Egon Börger, Gordon Fraser, and Uwe Glässer, for their time and dedication in reviewing this thesis and for their valuable feedback.

Contents

Chapter 1

Introduction

In several activities of our everyday life we get in touch, in a direct or indirect way, with hardware/software systems. In addition to the intentional use of PCs, laptops, or tablets, that is becoming common for work, study and for private entertainment, one can interact with HW/SW embedded systems also when she drives a car, takes a plane/high-speed train, pays with a credit card, or watches television (satellite or digital terrestrial); moreover, if she unfortunately suffers from an heart disease, she could have a pacemaker implanted, that is controlled by a specialized software. These are only some examples of HW/SW systems that are involved in our life, but the list could be very long; we could also mention systems that, although we do not directly use them, have a great impact on our society, as software systems that control nuclear plants, financial markets, space rockets, and so on.

Depending on the size and the importance of a system, its failure can constitute only an annoying inconvenience for the user (e.g., a bug in the software of the DVD player) or can have serious consequences for the manufacturer and/or the user. In literature, several systems failures have been described [16, 146]:

- One of the most famous is the explosion, 36 seconds after the lift-off, of the Ariane 5 rocket[1]; the amount lost was of half a billion dollars. The failure was caused by an uncaught exception, due to a floating-point error in a conversion from a 64-bit integer to a 16-bit signed integer.
- An error in the software of the baggage handling system delayed of 9 months the opening of the Denver airport, with a loss of 1.1 million dollars per day[2].
- Several HW/SW errors caused the recall of the faulty systems. Intel lost 475 million dollars for replacing Pentium II processors that had a faulty floting-point division unit. Also automotive industries had to recall some of their cars for defects in the onboard software: Toyota recalled some vehicles in 2010 for a bug in the anti-lock brake software[3], and Honda planned to recall 2.5 million vehicles for a bug in the transmission software[4].
- Sometimes systems failures have even worst consequences, since they can cause the loss of human life. For example, because of an error in the radiation therapy machine Therac-25, some patients were exposed to an overdose of radiation and six of them died[5].

It is apparent the need of having some techniques to produce systems that are as correct as possible. This is the main goal of *formal methods*, introduced in Section 1.1. Although applicable to the development of both hardware and software systems, from now on we only consider the latter ones.

[1]The report of the commission responsible for discovering the cause of the failure is reported at `http://www.esa.int/esaCP/Pr_33_1996_p_EN.html`

[2]`http://www.eis.mdx.ac.uk/research/SFC/Reports/TR2002-01.pdf`

[3]`http://en.wikipedia.org/wiki/2009-2011_Toyota_vehicle_recalls`

[4]`http://www.reuters.com/article/2011/08/05/us-honda-recall-idUSTRE77432120110805`

[5]`http://sunnyday.mit.edu/papers/therac.pdf`

1.1 Formal methods

To tackle the growing complexity of developing modern software systems that usually have embedded and distributed nature, and more and more involve safety critical aspects, *formal methods* (FMs) have been affirmed as an efficient approach to ensure the quality and correctness of the design, that permits to discover errors yet at the early stages of the system development.

Formal methods comprise all those notations and techniques that permit to describe and analyse systems in a formal way, i.e., relying on well founded mathematical theories, such as logic, automata theory or graph theory [146].

FMs provide several advantages when involved in software system engineering. According to [146], FMs can be used to achieve three main goals:

1. Having a *formal* description of the system under development. Since they are *formal*, FMs permit to describe, by means of *formal specification*s, the features and the expected behaviour of a system in an unambiguous way, so avoiding the misunderstanding that can arise if the system is described in natural language or using semi-formal design techniques as, for example, UML. Thanks to their unambiguous nature, formal specifications are also useful for system documentation: they can integrate less formal documentation, and taken as the final arbiter when the latter is ambiguous [33]. The use of a formal specification from the beginning of the system development permits to discover incompleteness, ambiguities and inconsistencies of the informal system requirements that, if discovered later during the development or testing, can cause a delay in the system deployment and an exceedance of the estimated costs.

2. Assisting the development of the system by providing a mechanism to obtain, in an automatic or semi-automatic way, the (partial) implementation of the system. *Refinement* approaches permit to obtain, starting from an abstract specification of the system, more and more detailed specifications, and possibly, in the end, the executable code. Each refinement step is usually well documented and proved correct, so that all design decisions are recorded and, if a re-design is required, can be examined again.

3. Catching and fixing design errors applying formal analyses methods that assure correctness w.r.t. the system requirements and guarantee the required system properties. To this purpose, validation (e.g., simulation, scenario-based validation and model review) and verification (e.g., model checking and theorem proving) techniques can be used.

These three points roughly correspond to the three *formalization levels* of a formal method (i.e., *formal specification*, *formal development/verification* and *machine-checked proofs*), identified in [33, 34].

There are several FMs available, each having its own strengths and weaknesses. Each FM is based on a different mathematical theory, provides a different notation, and is more suitable for achieving some goals rather than others. When choosing what FM to adopt, we should look for the FM that best addresses our necessities: we should define what formalization levels best fit our needs (not necessarily all three), and find a FM that covers the desired levels.

It is also possible to adopt different FMs to be used for different purposes during the development of the system. However, using more than a formal notation in the same project can have some disadvantages: it can be time consuming, since a formal specification of the system must be written for each FM adopted, and inconsistencies may arise between the different specifications. As suggested in [34], one viable solution is using *methods integration*, in which translations between different notations are provided, so that it is possible, having only one formal specification, to exploit the capabilities of different FMs.

Another factor that can influence the choice of a FM is its tool support. Even the best FM, without an appropriate set of tools that facilitate its usage, has few chances to be adopted [95]. In his proposal of the *Grand challenge project* [98] for the construction of a program verifier, Tony Hoare states that *development of the tools should consume a bit less than half the total effort.* The techniques presented in this text are all well tool supported, with the aim of making the use of FMs as easy as possible.

1.1.1 State-based formal methods

Among the several FMs available, some of them can be described as *state-based*, since they describe systems by using the notions of *state* and *transitions* between states. State-based FMs are sometimes preferred since they produce specifications that are more intuitive, since the notions of state and transition are close to the notions of program state and program execution that are familiar to any developer. Moreover, state-based FMs are usually executable and permit to be simulated, so having an abstraction of the execution of the system under development.

We can describe a generic state-based formal method as follows.

Definition 1.1 (State-based formal method)**.** *A generic* state-based formal method F *is described by the triple* $\langle S, S_0, T\rangle$*, being*

- S *the set of states;*
- S_0 *the set of initial states;*
- $T \subseteq S \times S$ *the transition relation.*

The previous definition can be used to *abstractly* describe several FMs but, since too generic, can not be used to *concretely* define any FM. In order to define a given state-based FM, Def. 1.1 must be extended by adding the distinguishing characteristics of the FM, as the way used to describe the states or the transition relation. Finite State Machines (FSMs), for example, add to the transitions some inputs, and to the states (or to the transitions) an output function. Kripke structures (see Section 3.1) have a labeling function that labels the states with the *facts* that are true in the state. NuSMV specifications (see Section 3.2) represent a particular class of Kripke structures, describing the states using a finite set of variables that range over finite domains, and defining the transition relation as a set of guarded updates of the variables. In Abstract State Machines (see Section 2.1) the states are first-order structures, and the transition relation is described by means of *transition rules.*

1.2 Motivations of the thesis

The aim of the thesis is to provide tool-assisted techniques that help the adoption of state-based FMs. In particular we address four main goals:

1. identifying a process for the development of an integrated framework around a formal method. The adoption of a formal method is often prevented by the lack of tools to support the user in the different development activities [95], as model editing, validation, verification, etc. Moreover, also when tools are available, they have usually been developed to target only one aspect of the system development process. So, having a well-engineered process that helps in the development of concrete notations and tools for a FM can make FMs of practical application. Indeed, as suggested by Parnas [145], FMs should not be an optional feature of the development process, but become an essential part of it; but, in order to achieve this result, they must prove to be really usable.

2. Promoting the integration of different FMs. As already said, having only one formal notation, for doing different formal activities during the development of the system, is preferable than having a different notation for each formal activity. Moreover such notation should be high-level: working with high level notations is definitely easier than working with low-level ones, and the produced specifications are usually more readable [125, 11]. This goal can be seen as a sub-goal of the first goal; indeed, in a framework around a formal method, it should also be possible to integrate other formal methods that better address some particular formal activities.

3. Helping the user in writing correct specifications. The basic assumption of any formal technique is that the specification, representing the desired properties of the system or the *model*[6] of the system, is correct. However, in case the specification is not correct, all the verification activities based on the specification produce results that are meaningless. So, validation techniques should assure that the specification reflects the intended requirements; besides traditional simulation (user-guided or scenario-based), also model review techniques, checking for common quality attributes that any specification should have, are a viable solution.

4. Reducing the distance between the formal specification and the actual implementation of the system. Several FMs work on a formal description of the system which is assumed to reflect the actual implementation; however, in practice, the formal specification and the actual implementation could be not conformant. A solution is to obtain the implementation, through refinements steps, from the formal specification, and proving that the refinements steps are correct. A different viable solution is to link the implementation with its formal specification and check, during the program execution, if they are conformant.

Moreover, we can identify two collateral goals of the previous four main goals.

First of all we should always give evidence that a proposed technique is effective, not only by some experiments, but by a formal evaluation of its strengths and weaknesses.

Second, we should always care about the performances of the proposed technique. Validation and verification techniques, in fact, usually have scalability problems due to their time and memory consumption.

Overview of the thesis

Part I describes three state-based formal notations, namely Abstract State Machines (ASMs) (Section 2.1), Kripke structures (Section 3.1), and NuSMV specifications (Section 3.2). A process for the development of a set of tools around a formal method is proposed in Section 2.2, where the process is instantiated for ASMs.

Part II, after a description of some validation techniques (Chapter 4) and an overview of *model review* techniques (Chapter 5), i.e., validation approaches for checking the *quality* of formal specifications, proposes a model review technique for NuSMV specifications and ASMs (Chapters 6 and 7). An approach for evaluating the fault detection capability of a model review technique is proposed in Chapter 8.

Part III introduces formal verification techniques. In Chapter 9, after a brief introduction to model checking, a technique for model checking ASMs is proposed (Section 9.2). Then, after an introduction to the notion of runtime monitoring (Chapter 10), Chapter 11 proposes a technique for checking at runtime the conformance of a Java program with its formal specification given in terms of ASMs. Chapter 12 proposes an approach for combining runtime monitoring with model-based testing with the aim of testing nondeterministic Java programs.

Finally, Part IV introduces some scalability issues that can arise in formal analysis techniques. Chapter 13 proposes a method to mitigate the state space explosion problem in test case generation using model checkers. Chapter 14, instead, proposes some optimizations for reducing the execution time of test case generation for boolean expression using SAT/SMT solvers.

A technique developed for a state-based FM usually can be (easily) adapted for being used with another state-based FM. This is the case of the two model review techniques for NuSMV and ASMs specifications, described in Chapters 6 and 7. In Chapter 8, instead, an approach for

[6]Specifying the system and specifying the properties of the system are two different activities [146]. With the term *formal specification*, depending on the contest, we can refer to the formal description of the system or to the properties that the system must assure; if there is no ambiguity, we use the term *formal specification* freely. When we do formal verification of a system (e.g., model checking), however, we usually have some properties that we want to prove over a formal description of the system: in this case the properties are the formal specification, whereas the description of the system is the *model* (i.e., the formal description assures the properties if it is a *model* for the properties, as defined in model theory).

discovering the equivalence between Kripke structures has been adapted to obtain a technique for checking the equivalence of NuSMV specifications, since the latter represent a particular class of the former. The runtime monitoring technique presented in Chapter 11 uses ASMs as formal specification notation, but any other state-based FM could be used.

Part I

State-based formal methods

Chapter 2

Abstract State Machines and ASMETA

2.1 Abstract State Machines (ASMs)

Abstract State Machines (ASMs) [32], previously known as Evolving Algebras [86], are a system engineering method that drives the development of systems (both software and hardware) from the requirements capture to the code implementation. ASMs have been successfully applied in different contexts: definition of industrial standards for programming and modeling languages, design and re-engineering of industrial control systems, modeling e-commerce and web services, design and analysis of protocols, architectural design, verification of compilation schema and compiler back-ends, etc.

ASMs permit to make requirements elicitation by constructing a *ground* model, i.e., a high-level description of the system under development. Ground models are formulated using an *application-oriented language which can be understood by all the stakeholders* [30]; so, thanks to their abstract and precise nature, ground models are system blueprints that can be used as *contracts* between the customer and the software designer. As stated in [30], ground models are *abstract yet complete*, that is they contain all the essential elements of the system under development (the interaction with the environment, the architectural system structure, etc.), but they do not model elements/behaviours that are not necessary for the overall understanding of the system. However, the points of the system that are left abstract, are properly marked in the ground model, so that it is clear what must be specified in later refinements. Although abstract, thanks to their mathematical nature, ground models can be already validated and verified. For example, since they are executable, they can be simulated to discover if they reflect the user expectations about the system.

Starting from the ground model, more detailed models can be iteratively obtained through the *refinement method*. Each refinement step consists in the implementation of some system requirements in a more detailed way. The notion of *correct refinement* between an ASM M and the refined ASM M^* – based on the equivalence of the runs of M and M^* – provides a theoretical support for the verification of the refinement correctness. The sequence of refinement steps is, in a traceable and documented way, a link between the abstract view of the system and the executable code.

ASMs dispel at least three of the myths about formal methods described in [88]:

5. *Formal methods involve complex mathematics and this makes it difficult their adoption.*

6. *Formal methods are incomprehensible to clients because of their mathematical nature.*

7. *Nobody uses formal methods for real projects.*

Myths 5 and 6 can be dispelled by the fact that ASMs can be viewed as *pseudo-code over abstract data* [32]: so their mathematical nature is somehow hidden to the software designer and to the customer that, usually, are both used to reading pseudo-code. Myth 7 is dispelled by the fact that ASMs have been successfully applied to real projects as, for example, a mathematical and an experimental analysis of Java and of the Java Virtual Machine (JVM) [158], or in

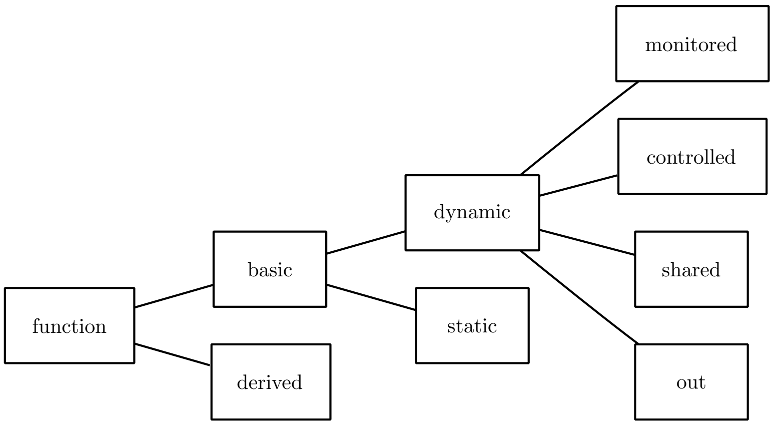


Figure 2.1: Classification of ASM functions

the specification of the Universal Plug and Play (UPnP) architecture for peer-to-peer network connectivity of intelligent devices [84].

ASMs are an extension of FSMs [30], where *states* are multi-sorted first-order structures, i.e., domains of objects with functions and predicates (treated as boolean functions) defined on them. The *transition relation* is specified by *rules* describing how functions interpretations change from one state to the next one.

2.1.1 Abstract states

Definition 2.1 (Signature)**.** *A* signature *(or* vocabulary*) is a finite set of function names, each having an* arity $n \in \mathbb{N}_0$ *that specifies the number of arguments of the function.*

Also if it is not specified, every signature contains the static constants *undef*, *true*, *false.*

Definition 2.2 (Location)**.** *The pair* $(f, (v_1, \dots, v_n))$ *composed by a function name* f *of arity* $n \geqslant 0$*, which is fixed by the signature, and an argument* $(v_1, \dots, v_n)$ *(empty if* $n = 0$*), which is formed by a list of dynamic parameter values* v_i *of whatever type, is called* location.

Locations can be viewed as an abstraction of memory units, in which the memory addressing and object referencing mechanisms are not specified [32].

Definition 2.3 (State)**.** *An ASM state* A *of a signature* Σ *is given by a non-empty set* X *(called the* superuniverse *of* A*) and interpretations for the function names in* Σ*.*

An interpretation for an n-ary function f, with $n > 0$, is a function $f^A \colon X^n \to X$. An interpretation c^a for a 0-ary function c is an element of X.

ASM functions can be partial, since they can be not totally defined. However, they can be seen as total functions, saying that the interpretation of a not specified location $f(v_1, \dots, v_n)$ is interpreted as the interpretation of the constant *undef* (i.e., $f^A(v_1, \dots, v_n) = \mathit{undef}^A$).

The superuniverse X is usually divided in smaller *universe*s (or *domain*s). A domain D can be described by a characteristic function g that indicates the elements of the superuniverse that constitute the domain, i.e., $\forall x \in X \;\; [x \in D \leftrightarrow g(x) = \mathit{true}]$.

2.1.1.1 Functions classification

ASMs functions are classified, as shown in Fig. 2.1, depending on the way in which they can be read and updated.

A first distinction is made between *basic* and *derived* functions. Basic functions are those that form the basic signature, whereas derived functions are those coming with a specification or computation mechanism given in terms of other basic functions.

Basic functions are further divided into *static*, which never change during any run of the machine, and *dynamic*, that may be changed by the environment or by machine *updates*. 0-ary static functions are also called *constant*s; 0-ary dynamic functions, instead, are also called *variable*s, since they can be viewed as the variables used in programming languages.

Dynamic functions are divided into:

- *monitored*: they are updated by the environment (or by another agent in case of a multi-agent machine), and can be only read by the machine (i.e., they can not appear in the left-hand side of an update rule); they identify the part of the dynamic state that is controlled by the environment;
- *controlled*: they can be read and updated by the machine (i.e., updated by transition rules), and they can not be updated by the environment (or other agents); they identify the part of the dynamic state that is directly controlled by the machine;
- *shared*: they can be read and updated both by the machine and the environment; they represent a combination of monitored and controlled functions; since they can be updated by multiple agents, usually a protocol is required to assure that the updates are consistent;
- *out*: they can be updated but not read by the machine (i.e., they can only appear in the left-hand side of an update rule), and read but not updated by the environment and other agents.

2.1.2 ASM transitions

The way in which an ASM changes its state, i.e., it changes the interpretation of its dynamic functions, is described by means of transition rules. Note that static functions never change their value. Derived functions, instead, although not directly modified by transition rules, can change their value among states since their computation mechanisms is based on the value of basic functions.

In Section 2.1.2.1 we give some basic definitions about the notion of state update, and in Section 2.1.2.2 we describe some transition rules that are used to define the transition relation.

2.1.2.1 ASM state update

Definition 2.4 (Update). *Location-value pairs* (loc, v) *(i.e.,* $(f, (v_1, \dots, v_n), v)$*) are called* up-date*s and represent the basic units of state change.*

Definition 2.5 (Update set). *The* update set *is the set of all the updates that can* fire *in a state.*

The updates that can fire are identified by the conditions imposed by the transition rules.

An update set can be applied to the machine only if it is *consistent*.

Definition 2.6 (Consistent update). *An update set updSet is* consistent *if it holds:*

$$(((f, (a_1, \dots, a_n)), b) \in \mathit{updSet} \wedge ((f, (a_1, \dots, a_n)), c) \in \mathit{updSet}) \rightarrow b = c$$

If the update set is consistent, it is not possible that a location is updated to two different values at the same time.

Definition 2.7 (Run). *A* run *(or* computation*) of an ASM is a finite or infinite sequence* $s_0, s_1, \dots, s_n, \dots$ *of states of the machine, where* s_0 *is an initial state and each* s_{i+1} *is obtained from* s_i *by simultaneously firing all of the transition rules which are enabled in* s_i.

If in a state the computed update set is not consistent, it is not applied and the run terminates.

Since ASMs can be nondeterministic, given an initial state, more than a run of length n can exist.

2.1.2.2 ASMs transition rules

In its simplest form, a transition rule has the form of *guarded update*

if *cond* **then** *updates*

where *cond* is a first-order formula without free variables, and *updates* a set of function updates of the form $f(t_1, \dots, t_n) := t$ which are simultaneously executed when *cond* is true. f is an arbitrary n-ary function and $t_1, \dots, t_n, t$ are first-order terms. To fire this rule in a state s_i, $i \geqslant 0$, all terms $t_1, \dots, t_n, t$ are evaluated at s_i to their values, say $v_1, \dots, v_n, v$, then the value of location $(f, (v_1, \dots, v_n))$ is updated to v, which represents the value of the location in the next state s_{i+1}.

Actually, ASMs provide a rich set of transition rules that, thanks to their high expressiveness, permit to describe complex guarded updates in a concise way.

We here briefly describe the transition rules available in the ASMs.

Skip rule It does not produce any effect.

skip

Update rule It updates the location $f(v_1, \dots, v_n)$ to the value v, being $v_1, \dots, v_n$ and v the evaluation of terms $t_1, \dots, t_n$ and t in the current state.

$$f(t_1, \dots, t_n) := t$$

Block rule It executes rules R_1 and R_2 simultaneously in parallel.

R_1 **par** R_2

Since R_1 and/or R_2 could be block rules as well, we introduce a more general notation that describes the parallel execution of n rules, with $n \geqslant 2$.

```
par
    $R_1$
    ...
    $R_n$
endpar
```

Conditional rule It executes rule R_{then} if the boolean condition *cond* is true, R_{else} otherwise[1].

```
if $cond$ then
    $R_{then}$
else
    $R_{else}$
endif
```

Forall rule It executes in parallel the rule R with all the values of x for which the boolean condition *cond* is true. The read-only variable x can occur both in the boolean condition and in the rule R.

```
forall $x$ in $D$ with $cond$ do
    $R$
```

[1]In concrete syntaxes, as AsmetaL (see Section 2.2) or the CoreASM syntax [68], the else branch is usually optional.

Choose rule It nondeterministically chooses a value for x for which the boolean condition *cond* is true and executes the rule R with the chosen value. If no value for x exists that satisfies the condition, nothing is done[2]. The read-only variable x can occur both in the boolean condition and in the rule R.

$$\begin{array}{l}\textbf{choose}\ x\ \textbf{in}\ D\ \textbf{with}\ \mathit{cond}\ \textbf{do}\\ \quad R\end{array}$$

Extend rule It takes a new element e from the *reserve* (i.e., a subset of the superuniverse X, containing fresh elements), it removes e from the reserve and adds it to the domain D. Then it executes rule R, where the new value e could be read.

$$\begin{array}{l}\textbf{extend}\ D\ \textbf{with}\ e\\ \quad R\end{array}$$

Let rule It associates to x the value of t and executes R. The read-only variable x can occur in the rule R.

$$\textbf{let}\ x = t\ \textbf{in}\ R$$

Call rule It calls the rule R using as parameters $t_1, \ldots, t_n$.

$$R(t_1, \ldots, t_n)$$

Turbo ASMs transition rules The transition rules described previously constitute the so called *basic* ASMs. A richer set of ASMs are the *turbo* ASMs that can contain transition rules for sequential composition, iteration, and the definition of parametrized submachines. The *sequential* rule ($R_1\,\textbf{seq}\,R_2$), for example, permits to decompose a step in *micro* steps that must be executed in sequence. We do not describe all the turbo rules here, since they are not used in the rest of the text.

2.1.3 Abstract State Machine

After having introduced the notion of signature, state, and transition rule, we can now formally define what is an *Abstract State Machine*, by first introducing the notion of *rule declaration.*

Definition 2.8 (Rule declaration)**.** *Given a transition rule R, containing occurrences of the free variables $x_1, \ldots, x_n$, a* rule declaration *for a rule name r of arity n is*

$$r(x_1, \ldots, x_n) = R$$

Given a rule call $r(t_1, \ldots, t_n)$, each occurrence of variable x_i in the body of R is substituted with t_i.

Definition 2.9 (Abstract State Machine)**.** *An* Abstract State Machine *is constituted by:*

1. *a signature Σ, together with a classification for all the functions;*
2. *a set of initial states S_0;*
3. *a set of rule declarations RD;*
4. *a* main rule *of arity 0 that acts as the starting point of the ASM and that invokes, directly or indirectly, the rules in RD.*

[2]Usually, in concrete syntaxes, the choose rule can also specify a rule R_{ifnone} that must be executed if no value for x satisfies the condition. The modified syntax is

$$\begin{array}{l}\textbf{choose}\ x\ \textbf{in}\ D\ \textbf{with}\ \mathit{cond}\ \textbf{do}\\ \quad R\\ \quad [\textbf{ifnone}\ R_{\mathit{ifnone}}]\end{array}$$

2.1.4 Multi-agent ASMs

In Section 2.1.3 we have assumed that an ASM describes the behaviour of a *single agent.*

However, ASMs also permit to specify *multi-agents* models, where different agents interact, each executing its own moves. In a multi-agent ASM, each agent specifies its own *program*, i.e., some basic/turbo rules that describe its behaviour.

Multi-agent ASMs can be *synchronous* or *asynchronous.*

In a synchronous multi-agent ASM all agents execute their programs in parallel, *synchronized using an implicit global system clock* [32]. A synchronous multi-agent ASM permits to decompose a complex single-agent ASM, identifying the part of the machine (signature and behaviours (i.e., transition rules)) that is due to a particular agent of the system under development.

In an asynchronous multi-agent ASM, instead, the moves of the agents can be scheduled in any desired order. A run of a multi-agent asynchronous ASM is a *partially ordered* set $(M, <)$ that guarantees three conditions:

1. *finite history*: each move $m \in M$ has finitely many predecessors;
2. *sequentiality of agents*: the set of moves of every agent is linearly ordered by $<$;
3. *coherence*: given an initial finite segment X of $(M, <)$, it exists a state $\sigma(X)$ that is the result of applying any maximal element $m \in M$ to state $\sigma(X - \{m\})$.

2.2 ASMETA: a toolset around ASMs

The practical integration of a formal method within a system development process is often prevented by the lack of tools supporting its use during the different development activities: model editing, simulation, validation, verification, tests generation, etc. Furthermore, when tools are available, it is often the case that they have been developed to cover well only one aspect of the whole system development process, while, at different steps, modelers and practitioners would like to switch tools to make the best of them while reusing information already entered about their models. Tools are usually loosely-coupled, they have their own notations and internal representation of models. This makes the integration of tools and the reuse of information hard to accomplish, so preventing a formal notation from being used in an efficient and tool supported manner during the entire software development life-cycle.

This was, for example, our experience with the ASMs. The increasing application of the ASM formal method for academic and industrial projects has caused a rapid development of tools around ASMs of various complexity and goals: tools for mechanically verifying properties using theorem provers or model checkers, and execution engines for simulation and testing. Since these ASM tools have been usually developed by different research groups, they are loosely coupled, and have syntaxes and internal representations of ASM models strictly depending on the implementation environment. This makes the encoding of ASM mathematical models not always natural and straightforward and makes the integration of these tools hard to accomplish.

In order to develop a set of integrated tools around a formal method, we present a process, based on the Model-Driven Engineering (MDE), which allows developing a family of tools supporting different activities of the development process, from system specification to system analysis, and that are strongly integrated in order to permit reusing information about models during several phases of the system life cycle (first aim of the thesis in Section 1.2). The process exploits MDE concepts and technologies, like metamodeling and automatic model-to-model and model-to-text transformation. It also facilitates software reuse, since several software artifacts are reused by all the tools, and it exploits several generation techniques and tools in order to automatically obtain several software artifacts starting from (meta)models.

The application of MDE principles in order to engineer software languages is well established in the context of domain-specific modeling languages [159]. It mainly consists of developing a model (called *metamodel*) to represent the modeling concepts of a language, their relationships, and their use and combination to build models (i.e., the abstract syntax of the language). We here propose to apply the same approach to formal notations as well. This (meta)modeling activity requires a certain effort and a deep understanding of the underlying formal notation. However, this effort is later compensated by the speed and the easiness with which software tools for the formal method can be developed. Indeed, one can automatically derive (through mappings or projections) from a metamodel-based abstract syntax, several different basic *artifacts* that can be reused. In particular, several language concrete notations and grammars can be easily derived or defined. They can be either human-comprehensible (textual and/or graphical) for editing models, and machine-comprehensible (like the XML Metadata Interchange format) for model handling by software applications. Software APIs for model representation in terms of programmable elements can also be easily obtained in a *generative* programming approach.

The proposed process is based on our experience in engineering a metamodel-based language for ASMs and in developing the ASMETA (ASM mETAmodeling) toolset [13, 81] which provides tools for developing, exchanging, and analysing ASM models. ASMETA is also a framework for developing new ASM tools and for integrating existing ones.

Although we here apply the proposed process to the ASMs, it is general enough and applicable to any kind of formal method: in [9], for example, it has been applied to the Finite State Machines (FSMs).

The remainder of this chapter is organized as follows. The fundamental concepts of the MDE approach for software development are briefly presented in Section 2.2.1. Section 2.2.2 presents the overall process of engineering a toolset around a formal method. Section 2.2.3 presents the application of the proposed approach to the ASMs: the result is the ASMETA framework, a set of tools around the ASMs.

2.2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) technologies, with a greater focus on architecture and automation, provide high levels of abstraction in software system development by promoting models as first-class artifacts to maintain, analyse, simulate, and eventually transform into code or into other models. Meta-modeling is a key concept of the MDE methodology and it is intended as a way to endow a language or a formalism with an abstract notation, so separating the abstract syntax and semantics of the language from its different concrete notations.

Although the foundations of the MDE as a paradigm are still evolving, some implementations of the MDE principles can be found in different meta-modeling/programming frameworks. The most commonly used are the OMG framework with the MOF (Meta Object Facility) as meta-language, the AMMA metamodeling platform with the KM3 meta-language, the Xactium XMF Mosaic initiative, the Software Factories and their Microsoft DSL Tools, the Model-integrated Computing (MIC), the Generic Modeling Environment for domain-specific modeling, the Eclipse Modeling Framework (EMF) with the Ecore meta-language, and the Eclipse subproject openArchitectureware.

The MDE methodology for engineering software languages is well established in the context of domain-specific languages [159]. The development process of a DSL consists, more or less, of the following four main activities:

1. defining the DSL core language model to reflect all relevant domain abstractions;
2. defining the behaviour of DSL language elements;
3. defining the DSL concrete syntax(es) by specifying symbols for language model elements and DSL production/composition rules;
4. integrating DSL artifacts with the platform/infrastructure by mapping the different artifacts to the target platform. This last activity produces transformations, integration tests, and platform extensions for the DSL.

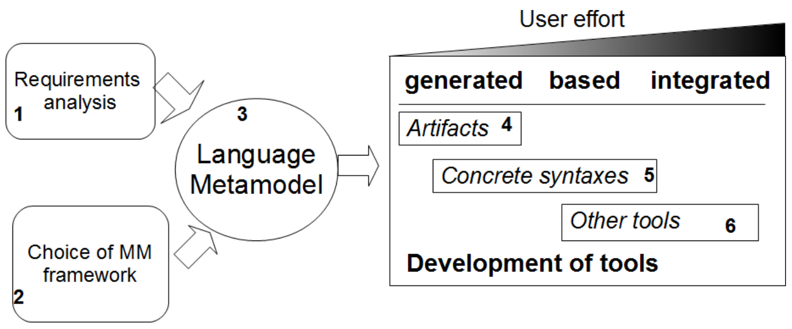


Figure 2.2: Model-Driven Process for Toolset Development

This model-driven development process can be adapted to a formal method with the overall goal of engineering a language and a set of integrated tools around it, as described in detail in Section 2.2.2.

2.2.2 Model-Driven process for toolset development

We here describe the steps of the model-driven process (see Fig. 2.2) a designer may undertake in order to engineer a set of integrated tools for a formal method, namely tools for model editing, exchange, validation, and verification. The process might require feedback loops and iterative development.

2.2.2.1 Requirements capture and analysis

During this step (1 in Fig. 2.2), concepts and constructs representing the expressive power of the formal method should be clearly pointed out. To this purpose, any official documentation should be taken in consideration, and, if language dialects already exist, it should be make clear if their characteristics have to be included in the new language.

2.2.2.2 Choice of a metamodeling framework and supporting technologies

The choice of a specific metamodeling framework (step 2 in Fig. 2.2) should not in principle prevent the use of models in other metamodeling frameworks, since model transformations (model-to-model, model-to-text, etc.) are supported by almost all metamodeling environments. In practice, metamodeling environments do not support all kinds of model transformations in the same way and problems may arise when changing technologies. Therefore, the choice of a metamodeling framework should consider the language artifacts one likes to generate from the metamodel. For example, if one is interested into a concrete textual notation, a framework supporting model-to-text transformations should be selected.

2.2.2.3 Design of a specification language for the formal method by metamodeling

During this step (3 in Fig. 2.2), the abstract syntax of a specification language is defined in terms of a *metamodel* describing the vocabulary of modeling concepts, the relationships existing among those concepts, and how they may be combined to create models. Possible constraints on the metamodel elements are expressed by the Object Constrain Language (OCL) [139]. Precise guide lines exist (e.g., [159]) to drive this modeling activity that leads to an instantiation of the chosen metamodeling framework for a specific domain of interest. This is a critical process step since the metamodel is the starting point for further tool development and remains the reference blueprint of the overall development process.

2.2.2.4 Development of tools

Software tools are developed starting from the language metamodel. They can be classified in *generated*, *based*, and *integrated*, depending on the decreasing use of generative technologies for their development. The effort required by the user increases, instead. Software tools automatically derived from the metamodel are considered generated. Based tools are those developed exploiting artifacts (APIs and other concrete syntaxes) and contain a considerable amount of code that has not been generated. Integrated tools are external and existing tools that are connected to the language artifacts: a tool may use just the XMI format, other tools may use the APIs or other derivatives. The difference between based and integrated tools is sometimes weak, and some tools can be can stay in both categories.

Development of *language artifacts* From the language metamodel, several *language artifacts* are generated (step 4 in Fig. 2.2) for model handling – i.e., model creation, storage, exchange, access, manipulatation – and these artifacts can be reused during the development of other applications. Artifacts are obtained by exploiting standard or proprietary mappings from the metamodeling framework to several technical spaces, as XML-ware for model serialization and interchange, and Java-ware for model representation in terms of programmable objects (through standard APIs).

Definition and validation of language *concrete syntax(es)* Language concrete notations (textual, graphical or both) can be introduced (step 5 in Fig. 2.2) for the human use of editing models conforming to the metamodel. Several tools exist to define (or derive) concrete textual grammars for metamodels. For example, EMFText [93] allows defining text syntax for languages described by an Ecore metamodel and it generates an ANTLR grammar file. TCS [107] (Textual Concrete Syntax) enables the specification of textual concrete syntaxes for Domain-Specific Languages (DSLs) by attaching syntactic information to metamodels written in KM3. A similar approach is followed by the TEF (Textual Editing Framework) [161]. Other tools, like the Xtext by openArchitectureWare (now integrated in the Eclipse Modeling Process) [171], following different approaches, may fit in our process as well. Depending on the degree of automation provided by the chosen framework, concrete syntax tools can be classified between generated and based software.

Once defined, concrete grammars must be also validated. To this aim, a pool of models written in the concrete syntax and acting as benchmark has to be selected. During this activity it is important to collect information about the coverage of language constructs (classes, attributes and relations of the language metamodel) to check that all of them are covered by the examples. Writing wrong models and checking that they are not accepted is important as well. Coverage evaluation can be performed by using a code coverage tool and instrumenting the parser accordingly. This validation activity is also useful to provide confidence that the metamodel correctly captures concepts and constructs of the underlying formal method.

2.2.2.5 Development of other tools

Metamodel, language artifacts, and concrete syntaxes are the foundations over which new tools can be developed and existing ones can be integrated (step 6 in Fig. 2.2).

2.2.3 ASMETA

We here report the experience of our research group[3] in engineering (over the last few years) a metamodel-based language and a toolset for the ASMs.

By following the steps of our model-driven design process, the ASMs have been provided with a set of tools, the ASMs mETAmodeling (ASMETA) toolset [13] (see Fig. 2.3), useful for the practical use of the ASMs in the systems development life-cycle. Concrete syntaxes have been defined, useful to create, store, access, validate, exchange and manipulate ASM models. Moreover, a general framework have been built, suitable for developing new ASM tools and for the integration of existing ones [81]. In the Fig. 2.3, the tools with a grey background are those that support the techniques for ASMs we propose in this thesis.

[3]Formal Methods and Software Engineering Laboratory (FM&SE Lab) – `http://fmse.di.unimi.it/`

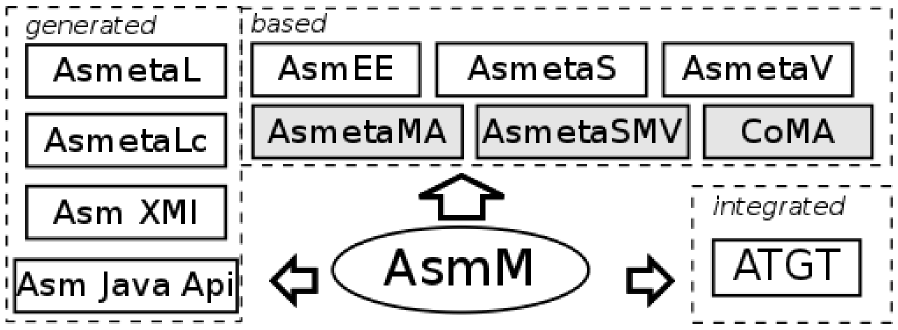


Figure 2.3: The ASMETA toolset

In the following we describe how the step described in Section 2.2.2 have been applied for the development of ASMETA.

Requirements capture and analysis

The initial developers of ASMETA started collecting all material available on the ASM theory and tool support. As official documentation about the ASM theory, they took [32], but they also considered to include constructs (i.e., particular forms of domains, special terms, derived rule schemes) from other existing notations (like XASM, ASM-WB, AsmGofer, and AsmL) for encoding ASM models.

Choice of a metamodeling framework and supporting technologies

As metamodeling framework, they initially chose the OMG MDA/MOF framework, the mainstream at the time the ASMETA project started. Later, they moved the ASMETA framework to the EMF-Ecore, open-source Eclipse framework, that is becoming the de-facto standard MDE platform, and it provides a great variety of supporting technologies and tools.

Design of a specification language for the formal method by metamodeling

The *Abstract State Machines Metamodel* (AsmM) [79] resulted into class diagrams representing all ASM concepts and constructs and their relationships. AsmM is available in both MDR/MOF and EMF/Ecore formats, but only the latter is actively maintained. The complete metamodel is organized in one package called `ASMETA` containing 115 classes, 114 associations, and 150 OCL class invariants, approximately.

Development of *language artifacts*

By exploiting projections from EMF to other technical spaces, from the AsmM they developed in a generative manner (see Fig.2.3): an *XMI* interchange format for ASMs, and Java *APIs* to represent ASMs in terms of Java objects. Both formalisms are useful to speed up the tooling activity around ASMs and, in fact, they are deeply used by several based tools of the framework.

Definition and validation of a language *concrete syntax*

In [78], the original developers of ASMETA defined general rules on how to derive a context-free EBNF grammar from a metamodel, and also provided guidance on how to automatically assemble a script file and give it as input to the JavaCC parser generator to generate a parser for the EBNF grammar of the textual notation. This parser is more than a grammar checker: it is able to process models conforming to their metamodel, to check for their well-formedness with respect to the OCL constraints of the metamodel, and to create instances of the metamodel through the use of the Java APIs.

Applying the technique explained in [78], they obtained *AsmetaL* [80], a platform-independent *textual notation*, to write ASM models. Table 2.1 shows the template of an ASM model (taken

AsmM elements	Concrete syntax
ASM	(**asm**|**module**) name
Header	[**import**] m_1 $[(id_{11} \dots id_{1h_1})]$ [**import**] m_k $[(id_{k1} \dots id_{kh_k})]$ [**export** $(id_1 \dots id_e)$ | **export** $*$] **signature**: $[dom\_decl_1 \dots dom\_decl_n]$ $[fun\_decl_1 \dots dom\_decl_m]$
where: - $(id_{i1} \dots id_{ih_i})$ are names for domains, functions and rules imported from module m_i (if omitted, all exported elements of m_i are imported) - $(id_1 \dots id_e)$ are names for exported domains, functions and rules (* to export all) - dom_decl_i and fun_decl_i are declarations of domains and functions	
Body	**definitions**: [**domain** $D_1 = DTerm_1$ ... **domain** $D_p = DTerm_p$] [**function** f_1 [(p_{11} **in** D_{11} ... p_{1h_1} **in** D_{1h_1})] $= FTerm_1$ ... **function** f_q [(p_{q1} **in** D_{q1} ... p_{qh_q} **in** D_{qh_q})] $= FTerm_q$] $[rule\_decl_1 \dots rule\_decl_r]$ $[invariant\_decl_1 \dots invariant\_decl_s]$
where: - $DTerm_i$ and $FTerm_i$ are terms defining domains D_i and functions f_i - p_{ij} are variables ranging in the domain D_{ij} and specifying the formal parameters of the function f_i - $rule\_decl_i$ and $invariant\_decl_i$ are declarations of rules and axioms	
Main rule	[**main** $rule\_decl$]
Initial state	[**default**] **init** sn: [**domain** $D_1 = DTerm_1$ ... $D_u = DTerm_u$] [**function** f_1 [(p_{11} **in** D_{11} ... p_{1h_1} **in** D_{1h_1})] $= FTerm_1$ ... **function** f_v [(p_{v1} **in** D_{v1} ... p_{vh_v} **in** D_{vh_v})] $= FTerm_v$] [**agent** A_1: $rule_1$ ... **agent** A_z: $rule_z$]
where: - sn is the name of the initial state - $DTerm_i$ and $FTerm_i$ are terms specifying the initial value of domains D_i and functions f_i - p_{ij} are variables ranging in the domain D_{ij} and specifying the formal parameters of the function f_i - A_i and $rule_i$ are the agents and their associated programs	

Table 2.1: Template of AsmetaL programs

from [80]); the complete documentation about the language can be found in [79, 12].

Moreover, they also, obtained *AsmetaLc*, a *text-to-model compiler*, to parse AsmetaL models and check for their consistency w.r.t. the AsmM OCL constraints. The AsmetaL and its compiler AsmetaLc can be considered in between generated and based tools, since they were partially derived from the metamodel (however we consider them based tools).

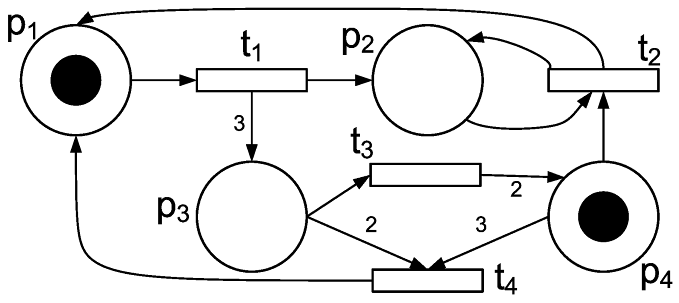


Figure 2.4: A Petri net

Example Code 2.1 shows an AsmetaL model that models the structure and the behaviour of the Petri net[4] shown in Fig. 2.4[5]. The abstract domains *Place* and *Transition* represent the places and the transitions of the net; their elements are defined as constants in the signature (i.e., $p_1 - p_4$, $t_1 - t_4$). The controlled function *tokens* stores the number of tokens of each place; the number of tokens can change during the evolution of the Petri net. The structure of the net is described by means of the static functions:

- *inArcWeight* contains the weights of the arcs between places and transitions; value 0 means that there is no arc between the place and the transition;
- *outArcWeight* contains the weights of the arcs between transitions and places; value 0 means that there is no arc between the transition and the place;
- *incidenceMatrix* contains the total number (positive or negative) of tokens that a place gains when a transition fires; it is the difference between the weight of the (possible) arc from the transition to the place, with the weight of the (possible) arc from the place to the transition[6].
- *isInputPlace* states if there is an arc between a place and a transition (i.e., if the weight of the possible arc is positive).

The derived function *isEnabled* states if a transition is enabled to fire: its interpretation depends on the number of tokens contained in the places connected to the transition through input arcs (input places).

The behaviour of a generic Petri net is given by the main rule that nondeterministically chooses a transition that is enabled and fires it with the macro call rule *r_fire*, that updates the marking of the places connected (with input and/or output arcs) with the chosen transition.

An invariant has been added to check that all the places do not assume a negative number of tokens.

[4]We briefly report the formal definition of Petri Nets [136]. An *infinite capacity* Petri net is a 5-tuple, $PN = (P, T, F, W, M_0)$:
- $P = \{p_1, p_2, \ldots, p_m\}$ is the set of places;
- $T = \{t_1, t_2, \ldots, t_n\}$ is the set of transitions;
- $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs;
- $W : F \rightarrow N^+$ is a weight function;
- $M_0 : P \rightarrow N$ is the initial marking;
- $P \cup T \neq \emptyset$: a net has at least a place or a transition;
- $P \cap T = \emptyset$: places and transitions are distinct.

In an execution step of a Petri Net, an *enabled* transition is nondeterministically chosen and fired. A transition is enabled if each place connected to the transition with an entering arc (i.e., belonging to $(P \times T)$) has at least as many tokens as the number specified by the label of the arc. When a transition fires, the tokens of the places connected with entering/exiting arcs are decreased/increased by the number specified by the labels of the arcs.

[5]The AsmetaL model is taken from [10], where we use ASMs to give semantics to Domain Specific Languages (DSLs) that are defined in terms of metamodels. In this particular example, ASMs provide the semantics for a DSL of Petri nets.

[6]Note that a place and a transition can be connected by an entering and an exiting arc at the same time. See, as an example, place p_2 and transition t_2 in Fig. 2.4.

Validation of AsmetaL By encoding a great number of ASM specifications, they validated the capability of AsmetaL to encode, in a natural and straightforward way, ASM mathematical models, and ASM specifications previously written in other ASM notations. The coverage of the metamodel was evaluated by instrumenting the AsmetaLc compiler with the EclEmma tool [62], assuring that all the metamodel constructs were covered at least once.

Development of other tools

In the ASMETA framework (see Fig.2.3), during the last years, several tools have been developed. The *based* tools are:

- the graphical front-end *ASMEE* (ASM Eclipse Environment), which acts as IDE and it is an Eclipse plug-in;
- the simulator *AsmetaS* [80] to execute ASM models (briefly presented in Section 4.1);
- the validator *AsmetaV* [39] with its language *Avalla* to express scenarios, for scenario-based validation of ASM models (briefly presented in Section 4.2);
- the model reviewer *AsmetaMA*, to check for common quality attributes of ASM models (proposed in Chapter 7);
- the model checker *AsmetaSMV* [6] for model verification by NuSMV (proposed in Chapter 9.2);
- the runtime monitoring tool *CoMA* for checking at runtime the conformance of a Java code with its ASM formal specification (proposed in Chapters 11 and 12).

Another tool, instead, is classified as *integrated*, since it was developed independently and later added to the ASMETA framework: the *ATGT* tool [77] is an ASM-based test case generator, based upon the SPIN model checker, that accepts AsmetaL as input notation.

```
asm petriNet
import StandardLibrary

signature:
    abstract domain Place
    abstract domain Transition

    controlled tokens : Place −> Integer
    static inArcWeight: Prod(Place, Transition) −> Integer
    static outArcWeight: Prod(Transition, Place) −> Integer
    static incidenceMatrix: Prod(Place, Transition) −> Integer //tokens gained after a transition fire
    static isInputPlace: Prod(Place, Transition) −> Boolean
    static p1: Place
    static p2: Place
    static p3: Place
    static p4: Place
    static t1: Transition
    static t2: Transition
    static t3: Transition
    static t4: Transition
    derived isEnabled : Transition −> Boolean

definitions:
    function inArcWeight($p in Place, $t in Transition) =
        switch($p, $t)
            case (p1, t1): 1
            case (p2, t2): 1
            case (p3, t3): 1
            case (p3, t4): 2
            case (p4, t2): 1
            case (p4, t4): 3
            otherwise 0
        endswitch

    function outArcWeight($t in Transition, $p in Place) =
        switch($t, $p)
            case (t1, p2): 1
            case (t1, p3): 3
            case (t2, p1): 1
            case (t2, p2): 1
            case (t3, p4): 2
            case (t4, p1): 1
            otherwise 0
        endswitch

    function incidenceMatrix($p in Place, $t in Transition) = outArcWeight($t, $p) − inArcWeight($p, $t)

    function isInputPlace($p in Place, $t in Transition) = inArcWeight($p, $t) > 0

    function isEnabled ($t in Transition) =
        (forall $p in Place with isInputPlace($p, $t) implies tokens($p) >= inArcWeight($p, $t))

    rule r_fire ($t in Transition) =
        forall $p in Place with true do
            tokens($p) := tokens($p) + incidenceMatrix($p, $t)

    invariant over tokens: (forall $p in Place with tokens($p) >= 0)

    main rule r_Main =
        choose $t in Transition with isEnabled($t) do
            r_fire [$t]

default init s0:
    function tokens($p in Place) = at({p1 −> 1, p2 −> 0, p3 −> 0, p4 −> 1}, $p) //initial marking
```

Code 2.1: ASM model of the Petri net shown in Fig. 2.4

Chapter 3

Kripke structures and the NuSMV model checker

3.1 Kripke structures

Kripke structures were introduced by Kripke to represents models of basic modal logic [100], i.e., propositional logics extended with the connectives $\Box$ and $\Diamond$ that express two modalities of truth, *necessity* and *possibility*. Kripke structures use states to represent *worlds* in which some facts (atomic propositions) are true, and transitions between states to show how the worlds are connected. Checking that a fact p is necessarily true (i.e., $\Box p$) means checking that p holds in all the reachable states. Checking that a fact p is possibly true (i.e., $\Diamond p$) means checking that it exists a reachable state in which p holds.

We here consider Kripke structures as models of a temporal logic, since temporal logics are special kinds of modal logic.

In Section 3.1.1 we give some basic definitions about Kripke structures. In Section 3.1.2 we present a theorem about the equivalence of Kripke structures: it will be used in Chapter 8 to detect equivalence between NuSMV models, since they are a form of Kripke structures.

3.1.1 Definitions

Definition 3.1 (Kripke structure)**.** *A* Kripke structure *is a quadruple* $M = \langle S, S^0, T, \mathcal{L}\rangle$ *where*

- S *is a set of states;*
- $(S^0 \subseteq S) \neq \varnothing$ *is the set of initial states;*
- $T \subseteq S \times S$ *is the transition relation;*
- $\mathcal{L} : S \to \mathcal{P}(AP)$ *is the proposition labeling function, where* AP *is a set of atomic propositions.*

Definition 3.2 (Finite Kripke structure)**.** *A Kripke structure where* S *and* AP *are finite.*

Definition 3.3 (Total Kripke structure)**.** *A Kripke structure with a left-total transition relation, i.e.,*

$$\forall s \in S, \exists s' \in S \colon (s, s') \in T$$

A not total structure is said partial.

Fig. 3.1 shows the graphical representation of the finite and total Kripke structure $K = \langle\ \{s_1, s_2, s_3, s_4, s_5\}, \{s_1, s_2\}, \{(s_1, s_2), (s_1, s_5), (s_2, s_3), (s_2, s_5), (s_3, s_4), (s_4, s_3), (s_5, s_1)\}, \mathcal{L}_K\rangle$ where $\mathcal{L}_K$ is defined as follows

$\mathbf{s}$	s_1	s_2	s_3	s_4	s_5
$\mathcal{L}_\mathbf{K}(\mathbf{s})$	$\{p\}$	$\{p, q\}$	$\{p\}$	$\{r\}$	$\{r, t\}$

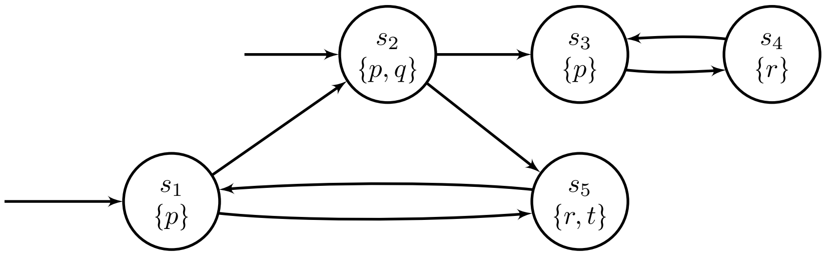


Figure 3.1: Example of a finite and total Kripke structure

Definition 3.4 (Computation tree). *Given a Kripke structure $K = (S, S^0, T, \mathcal{L})$, a computation tree of M is a tree structure where the root is an initial state $s_0 \in S^0$, and the children of a node $s \in S$ in the computation tree are all the states $s' \in S$ such that there exists a transition $(s, s') \in T$.*

If the Kripke structure is total, the computation tree is *infinite*, i.e., it has no leaves.
Fig. 3.2 represents the computation tree of the Kripke structure shown in Fig. 3.1.

Definition 3.5 (Path). *A* finite path *is a sequence of states in S*

$$\pi = s_1, s_2, \ldots, s_n$$

such that $\forall i \in [1, n-1]\colon (s_i, s_{i+1}) \in T$. The length *of the path, denoted by $|\pi|$, is n.*
An infinite path *is a sequence of states in S*

$$\pi = s_1, s_2, \ldots$$

such that $\forall i \geqslant 1\colon (s_i, s_{i+1}) \in T$. The length *of the path is ∞.*

With π_i we identify the i-th state of π (i.e., $\pi_i = s_i$). With π^i we identify the tail of the path starting in π_i (i.e., $\pi^i = s_i, s_{i+1}, \ldots$).
Π is the set of all the paths in K.
$\Pi^0 \subseteq \Pi$ is the set of all the paths such that the starting state $s_1 \in S^0$.

Definition 3.6 (Reachability). *A state $s \in S$ is reachable in K if there exists a finite path $\pi^0 = s_1, \ldots, s_n \in \Pi^0$ such that $s_n = s$, i.e.,*

$$\mathit{isReach}(s) \triangleq \exists \pi^0 = s_1, \ldots, s_n \in \Pi^0\colon s_n = s$$

We denote by $\mathit{reach}(K) \subseteq S$ the set of reachable states of the structure K.

Definition 3.7 (Successor state). *A state s' is a* successor *of another state s if $(s, s') \in T$. We denote by $\mathit{next}(s)$ the set of the successor states of s, i.e.,*

$$\mathit{next}(s) = \{s' \in S\colon (s, s') \in T\}$$

3.1.2 Equivalence for Kripke structures

We here report the essential definitions and theorems on the equivalence of Kripke structures [36, 15].

Let $K_1 = \langle S_1, S_1^0, T_1, \mathcal{L}_1\rangle$ and $K_2 = \langle S_2, S_2^0, T_2, \mathcal{L}_2\rangle$ be two total and finite Kripke structures with the same set of atomic propositions AP. A relation E can be defined on $S_1 \times S_2$ to express the equivalence between states of the two structures K_1 and K_2; two states are equivalent if they have the same labels and bring to next states having the same labels.

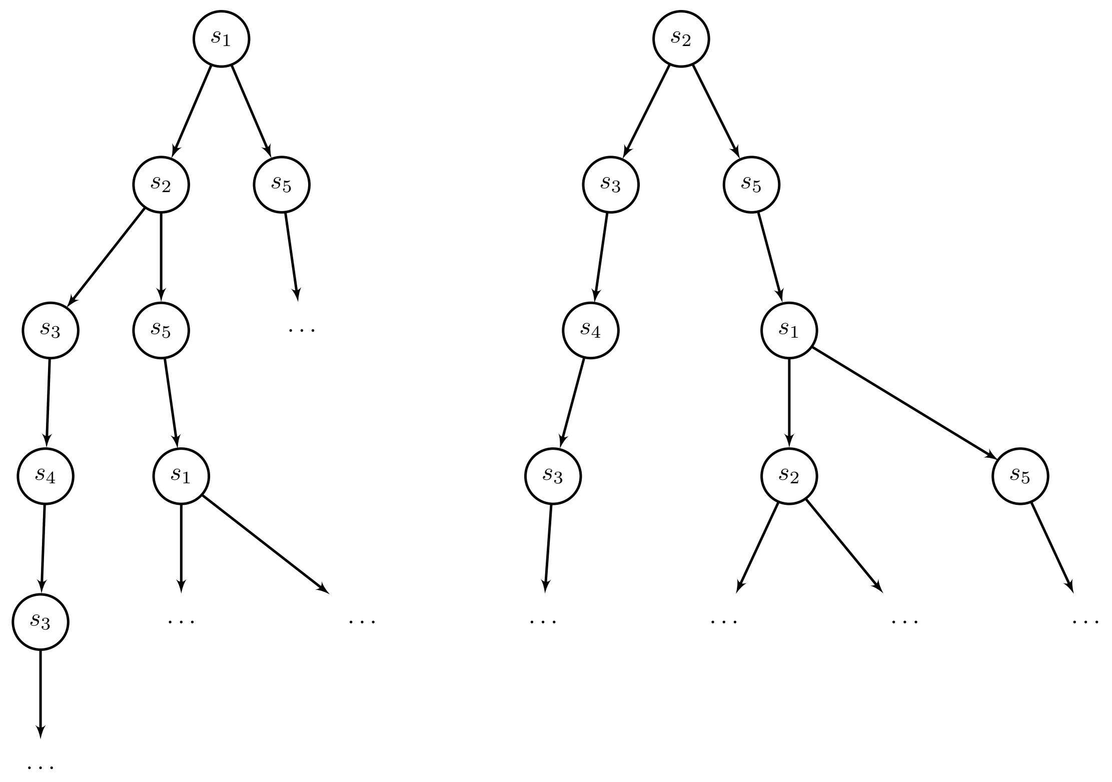


Figure 3.2: Computation trees of the Kripke structure in Fig. 3.1

Definition 3.8 (State equivalence). *$\forall s_1 \in S_1 \forall s_2 \in S_2$ we say $s_1 E s_2$ iff the following condition holds:*

$$\begin{array}{ll} \mathcal{L}_1(s_1) = \mathcal{L}_2(s_2) & \wedge \\ \forall s_1' \in next(s_1)\ \exists s_2' \in next(s_2) : \mathcal{L}_1(s_1') = \mathcal{L}_2(s_2') & \wedge \\ \forall s_2' \in next(s_2)\ \exists s_1' \in next(s_1) : \mathcal{L}_2(s_2') = \mathcal{L}_1(s_1') & \end{array}$$

Theorem 3.1 (Structure equivalence). *Let K_1 and K_2 be two Kripke structures with the same set of atomic propositions. If the following properties hold (initial states have same labeling and reachable states are equivalent):*

$$\forall s_1^0 \in S_1^0, \exists s_2^0 \in S_2^0 \colon \left[\mathcal{L}_1(s_1^0) = \mathcal{L}_2(s_2^0)\right] \tag{3.1}$$

$$\forall s_2^0 \in S_2^0, \exists s_1^0 \in S_1^0 \colon \left[\mathcal{L}_2(s_2^0) = \mathcal{L}_1(s_1^0)\right] \tag{3.2}$$

$$\forall s_1 \in reach(K_1)\ \exists s_2 \in reach(K_2) \colon s_1\ E\ s_2 \tag{3.3}$$

$$\forall s_2 \in reach(K_2)\ \exists s_1 \in reach(K_1) \colon s_2\ E\ s_1 \tag{3.4}$$

then K_1 and K_2 are equivalent.

3.2 The NuSMV model checker

NuSMV [48, 137] (New Symbolic Model Verifier) is known as a model checker derived from the CMU SMV [129]. It allows for the representation of synchronous and asynchronous finite state systems, and for the analysis of specifications expressed in *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL), using Binary Decision Diagrams (BDD)-based and SAT-based model checking techniques. Heuristics are available for achieving efficiency and partially controlling the state space explosion problem.

For an introduction to the model checking problem and to CTL and LTL, we remind to Section 9.1. Here, we are mainly interested in describing the syntax of the model checker NuSMV (Section 3.2.1), and how a NuSMV model describes a Kripke structure (Section 3.2.2).

3.2.1 NuSMV syntax

A NuSMV model describes the behaviour of a Kripke structure in terms of a *possible next state* relation between states that are determined by the values of variables. Transitions between states are determined by the updates of the variables. According to the model *operational* description, a NuSMV model is made of two principal sections:

- **VAR** that contains variable declarations. A variable type can be **boolean**, integer defined over intervals or sets, an enumeration of symbolic constants, or a bounded array of one of the three previous types.
- **ASSIGN** that contains the initialisation (by the instruction **init**) and the update mechanism (by the instruction **next**) of variables. It is also possible to define the value of a variable in the current state, rather than defining its initial value and its transition relation; in this case the init and next assignments can not be declared.

Moreover, a **DEFINE** statement can be used as a macro to syntactically replace an *identifier* with the *expression* it is associated with. The associated expression is always evaluated in the context of the statement where the identifier is declared. Note that a DEFINE identifier does not introduce a new variable and so it does not increase the state space.

The properties specification can be done in the **CTLSPEC** (resp. **LTLSPEC**) section, that contains the CTL (resp. LTL) properties to be verified. It is possible to name a property, specifying an identifier after the keyword *NAME*. Naming the properties permits to ask the model checker to check only a particular property.

A *state* of the model is an assignment of values to variables/definitions. According to the NuSMV language definition, there exist the following four kind of assignments:

- *simple* assignment for defining the value of a variable *var* in the *current* state:

```
ASSIGN
    var := simple_expression;
```

- *init* assignment for defining the value of a variable *var* in the *initial* state:

```
ASSIGN
    init(var) := simple_expression;
```

- *simple* assignment for defining the value of a variable *var* in the *next* state:

```
ASSIGN
    next(var) := next_expression;
```

- *macro definition* of an identifier *def*:

```
DEFINE
    def := simple_expression;
```

In simple/init assignments and macro definitions, only *simple_expressions* can be used, i.e., expressions built from the values of variables in the current state. In next assignments, instead, it is possible to use *next_expressions*, i.e., expressions in which also the values of other variables in the next state can be checked (through the **next** operator that can access the value of a variable in its next state). The *circular dependency rule* requires that there are no cycles in the variables dependency graph. For example, in the following model, the definition of the next values of variables *a*, *b* and *c* contains a cycle.

```
MODULE main
  VAR
    a: boolean;
    b: boolean;
    c: boolean;
  ASSIGN
    next(a) := next(b);
    next(b) := next(c);
    next(c) := next(a);
```

In both *simple_expression* and *next_expressions*, a variable value can be determined either unconditionally or conditionally, depending on the form of the expression.

Conditional expressions can be:

1. An **if-then-else** expression

   ```
   cond1 ? exp1 : exp2
   ```

   which evaluates to *exp1* if the condition *cond1* evaluates to true, and to *exp2* otherwise.

2. A **case** expression:

   ```
   case
     left_expression_1 : right_expression_1 ;
     ...
     left_expression_n : right_expression_n ;
   esac
   ```

   which returns the value of the first *right_expression_i* such that the corresponding *left_expression_i* condition evaluates to true, and the previous i-1 left expressions evaluate to false. The type of the expressions on the left-hand side must be boolean. An error occurs if all expressions on the left-hand side evaluate to false. To avoid these kinds of errors, NuSMV performs a static analysis and, if it believes that in some states no left expression may be true, it forces the user to add a *default case* with *left_expression* equal to TRUE. This kind of analysis is conservative: sometimes the user must add a default case even if it is not necessary. For example, in the following model, in the case expression used in the definition of the next value of variable x, the first two left expressions are complete since x never takes value 2; however, NuSMV requires to add the default condition.

   ```
   MODULE main
     VAR
       x: 1..3;
     ASSIGN
       init(x) := 1;
       next(x) :=
         case
           x = 1: 3;
           x = 3: 1;
           TRUE: 2;-- default condition useless, but still required
         esac;
   ```

In NuSMV it is possible to model *nondeterministic behaviours* by

(a) not assigning any value to a variable that, in this case, can assume any value of its finite domain;

(b) assigning to a variable a value randomly chosen from a set.

The behaviour of a variable can be always nondeterministic, in case both its initialisation and the definition of its transition relation are always nondeterministic, or be nondeterministic only in some states. In case of variable nondeterministic definition, NuSMV creates as many states

```
MODULE main
    VAR
        req: boolean;
        state: {ready, busy};
    ASSIGN
        init(state) := ready;
        next(state) :=
            case
                state = ready & req: busy;
                TRUE: {ready, busy};
            esac;
CTLSPEC AG(req -> AF state = busy)
```

Code 3.1: Example of NuSMV model

```
MODULE main
    VAR
        req: boolean;
        state: {ready, busy};
    INIT state = ready;
    TRANS (state = ready & req) -> next(state) = busy;

CTLSPEC AG(req -> AF state = busy)
```

Code 3.2: Declarative version of the model shown in Code 3.1

as the number of possible values; if there is no definition at all, all the values of the variable type are considered.

Code 3.1 shows a simple NuSMV model, taken from the documentation of NuSMV [42], that abstractly models the behaviour of a system that provides a service. The states of the model are identified by the values of the boolean variable *req*, that signals if there is a request to be satisfied, and the enumerative variable *state* that indicates if the system is *busy* (in satisfying a request or in other activities), or if it is *ready* for receiving a request. Since the initial and next values of variable *req* are not defined, they are always fixed nondeterministically. The *state* of the system is *ready* in the initial state; if there is a request and the system is *ready*, the system becomes *busy* (i.e., it starts fulfilling the request), otherwise it nondeterministically decides to be *ready* or *busy* (note that it can be busy also if there is no request to fulfil).

Definition in terms of propositional formulae NuSMV offers a more *declarative* way of defining initial states and transition relations, directly in terms of propositional formulae. Initial states can be defined by the keyword **INIT** followed by a boolean expression that describes constraints that must be satisfied in the initial states. Transition relations can be expressed through the keyword **TRANS** followed by a boolean expression describing the relation between the current and the next state. Invariant conditions can be expressed by the command **INVAR** followed by a boolean expression that must be true in each state. **INIT** and **INVAR** sections can contain only simple-expressions, whereas the **TRANS** section can contain also next-expressions in order to describe variables updates.

Code 3.2 shows a NuSMV model in which we describe, in a declarative way, the same system described by the model shown in Code 3.1. There is a technique [41] that permits to automatize the translation between an operational model to its declarative description; it is based on the following equivalences.

```
ASSIGN a := exp;        is equivalent to    INVAR a in exp;
ASSIGN init(a) := exp;  is equivalent to    INIT a in exp;
ASSIGN next(a) := exp;  is equivalent to    TRANS next(a) in exp;
```

```
MODULE philosopher(leftFork, rightFork)
  VAR
    status: {EATING, THINKING};
  ASSIGN
    init(status) := THINKING;
    next(status) :=
      case
        (status=THINKING & leftFork & rightFork) |
            status=EATING: {EATING, THINKING};
        TRUE: THINKING;
      esac;
    next(leftFork) :=
      case
        next(status)=EATING: FALSE;
        status=EATING: TRUE;
        TRUE: leftFork;
      esac;
    next(rightFork) :=
      case
        next(status)=EATING: FALSE;
        status=EATING: TRUE;
        TRUE: rightFork;
      esac;

JUSTICE running --the process is executed infinitely often
--liveness properties (always possible to eat and think)
CTLSPEC AG(EF(status=EATING))
CTLSPEC AG(EF(status=THINKING))
```

```
MODULE main
  VAR
    fork1: boolean; -- TRUE if free
    fork2: boolean;
    fork3: boolean;
    fork4: boolean;
    fork5: boolean;
    philo1: process philosopher(fork1, fork2);
    philo2: process philosopher(fork2, fork3);
    philo3: process philosopher(fork3, fork4);
    philo4: process philosopher(fork4, fork5);
    philo5: process philosopher(fork5, fork1);
  ASSIGN
    init(fork1) := TRUE;
    init(fork2) := TRUE;
    init(fork3) := TRUE;
    init(fork4) := TRUE;
    init(fork5) := TRUE;
```

Code 3.3: NuSMV model of the dining philosophers problem

Synchronous/asynchronous systems A NuSMV model can be decomposed in modules; at least the module *main* is always present. Modules can also have parameters that are passed by reference.

New instances of a module can be created in the VAR section of a different module. Modules can be instantiated in a synchronous or asynchronous way (using the keyword *process*). Synchronous instances are executed together in parallel. Asynchronous instances are called *processes*; at each step, one process is nondeterministically chosen to be executed. Processes have a special boolean variable *running* that signals if a process is in execution.

Code 3.3 shows the NuSMV model of the dining philosophers problem in which five processes of the module *philosopher* are created in the main module, each taking as actual parameters the left and the right fork. The *justice* constraint[1] requires to restrict the attention only to execution paths in which variables *running* of all the instances of *philosopher* are true *infinitely often*, i.e., those paths in which the instances are executed infinitely often.

3.2.2 Relation between NuSMV models and Kripke structures

We here describe the relation between NuSMV models and Kripke structures. In Section 3.2.2.1 we see how it is possible to derive a total and finite Kripke structure starting from a NuSMV model, and in Section 3.2.2.2 the other way round.

[1]Fairness constraints restrict the attention of the model checker only to *fair* execution paths. A *justice* constraint (keyword **JUSTICE** or **FAIRNESS**) requires that a boolean formula is true infinitely often. A *compassion* constraint (keyword **COMPASSION**) requires that, given two boolean formulae (p, q), if p is true infinitely often in a fair path, then also q is true infinitely often in the same path.

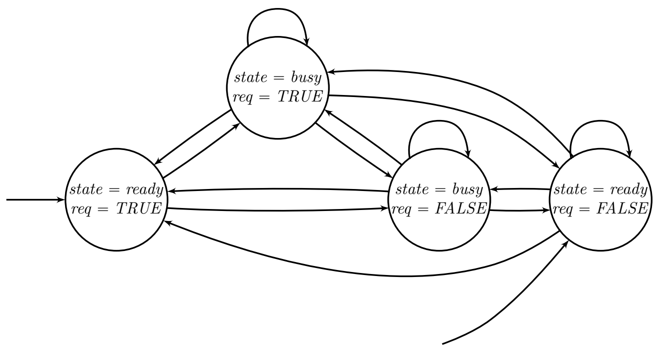


Figure 3.3: Kripke structure of the NuSMV model shown in Code 3.1

3.2.2.1 Deriving a Kripke structure from a NuSMV model

A NuSMV model represents a total and finite Kripke structure (see Defs. 3.1, 3.2 and 3.3) by means of a set of variables, describing how they modify their values among states. Let's give some definitions for describing this relationship.

Definition 3.9 (NuSMV model)**.** *Given a NuSMV model M, we identify with $var(M) = \{v_1, \dots, v_n\}$ the finite fixed set of its variables taking values over domains $D_1, \dots, D_n$. The model is defined by the triple $M = \langle S, S^0, T\rangle$, where:*

- *S is a finite set of states; each state is uniquely identified by the value of the variables in the state, i.e.,*

$$\forall s_1, s_2 \in S \left[\left(\bigwedge_{i=1}^{n} [\![v_i]\!]_{s_1} = [\![v_i]\!]_{s_2} \right) \leftrightarrow s_1 = s_2 \right]$$

  There are $\prod_{i=1}^{n} |D_i|$ states (not necessarily all reachable);

- *$S^0 \subseteq S$ is the set of initial states;*

- *T is the transition relation defined through the updates of the state variables.*

Definition 3.10 (NuSMV model as Kripke structure)**.** *A NuSMV model $M = \langle S, S^0, T\rangle$ represents a Kripke stucture $K = \langle S, S^0, T, \mathcal{L}\rangle$ where:*

- *the set of atomic propositions AP is composed by the equalities $v_i = d_i$ (with $v_i \in var(M)$ and $d_i \in D_i$). The number of atomic propositions is $\sum_{i=1}^{n} |D_i|$;*

- *each state s is labeled by n atomic propositions, one for each variable:*

$$\mathcal{L}(s) = \{v_1 = d_1, \dots, v_n = d_n\}$$

From Def. 3.10 it is possible to easily define a procedure for deriving a Kripke structure from a NuSMV model. Note that the produced Kripke structure is finite, since the number of states is finite, and the labeling function is injective, since different states have different labels. Fig. 3.3 shows the corresponding Kripke structure of the NuSMV model shown in Code 3.1.

```
MODULE main
  VAR
    state: {s1, s2, s3, s4, s5};
  DEFINE
    p := state in {s1, s2, s3};
    q := state = s2;
    r := state in {s4, s5};
    t := state = s5;
  ASSIGN
    init(state) := {s1, s2};
    next(state) :=
      case
        state = s1: {s2, s5};
        state = s2: {s3, s5};
        state = s3: s4;
        state = s4: s3;
        state = s5: s1;
        TRUE: state;
      esac;

CTLSPEC AF r
```

Code 3.4: NuSMV model derived from the Kripke structure shown in Fig. 3.1

3.2.2.2 Deriving a NuSMV model from a Kripke structure

It is also possible to derive a NuSMV model starting from a general Kripke structure in which the atomic predicates are general (i.e., they are not equalities between variables and values). The only constraint is that the Kripke structure must be finite (see Def. 3.2) and total (see Def. 3.3).

Given a finite and total Kripke structure $K = \langle S, S^0, T, \mathcal{L}\rangle$, a NuSMV model can be obtained in the following way:

1. declare an enumerative variable *state*, taking as values all the identifiers of the states;
2. define, for each atomic predicate $p \in AP$, a definition in the DEFINE section as follows

   ```
   DEFINE p := state in {s_1, ..., s_k};
   ```

   where *s_1, ..., s_k* are all the states s_j that are labeled by *p*, i.e., $p \in \mathcal{L}(s_j)$.
3. define the initialisation of the variable *state* as follows

   ```
   ASSIGN init(state) := {s_1, ..., s_r};
   ```

   where $\{s\_1, \dots, s\_r\} = S^0$.
4. define the next value of the variable *state* as follows

   ```
   ASSIGN next(state) :=
            case
              state = s_1: nextStates(s_1);
              ...
              state = s_n: nextStates(s_n);
              TRUE: state;
            esac;
   ```

   where $nextStates(s\_i) = \{s\_1, \dots, s\_t\}$ are all the next state of s_i.

Code 3.4 shows the NuSMV model derived from the Kripke structure shown in Fig. 3.1; a CTL property has been added later to the obtained model. Note that the atomic predicates are not used to define the transition relation of the model, but only in the temporal properties.

Part II

Validation

Chapter 4

Model validation

A definition given by Boehm of *system validation* is that it consists of all those techniques that permit to answer the question “Are we building the right product?” [29].

We here only consider *model validation*, i.e., the process of investigating a model (intended as formal specification) with respect to the user perceptions, in order to ensure that the specification really reflects the user needs and statements about the application, and to detect faults in the specification as early as possible with limited effort. Model validation can give us enough confidence that the formal specification correctly describes the intended requirements of the application under development and, so, can be used for other activities like formal verification, or it can be refined into a more detailed specification. Note that the validation of the model is a paramount activity, since using a *wrong* model can have serious consequences in terms of costs overrun and delay in the system deployment. Indeed, if the wrong requirements described by the model are implemented in the code, finding and fixing the faults in the code is definitely more difficult and expensive than correcting the model.

Some validation techniques are:

- *simulation*: the user executes the model by providing certain inputs, and she observes if the outputs are the expected ones or not.
- *scenarios generation*: the user builds scenarios describing the expected behaviour of a system. A scenario describes a simulation of the model, providing a series of inputs to exercise the model, and checks that the output is as expected. It is a more automated version of the basic simulation.
- *model review*: models are (automatically) examined to determine if they have some quality attributes that any model of a particular notation should have. We remind to Chapter 5 for an introduction to model review, and to Chapters 6 and 7 for the description of two model review techniques we propose for NuSMV and ASMs specifications.

We here briefly describe simulation and scenarios generation, as implemented in the ASMETA framework (see Section 2.2). In Section 4.1 we see the ASMs simulator, and in Section 4.2 the scenario-based validator. We introduce these two techniques since they have been used in some of the works described in this thesis. The simulator is used by CoMA, the runtime conformance checker of Java code, described in Chapter 11; the scenario-based validator, instead, can be used to reproduce the counterexamples returned by the model checker AsmetaSMV, described in Section 9.2.

4.1 ASMs simulator

Simple model validation can be performed by basic simulation, in which the user can get a general idea of the behaviour of the written specification and determine if it reflects her expectations. Fixing the errors found at this stage of development has nearly no cost, but it produces great benefits.

```
<State 1 (controlled)>
Place={p1,p2,p3,p4}
Transition={t1,t2,t3,t4}
tokens(p1)=0
tokens(p2)=1
tokens(p3)=3
tokens(p4)=1
</State 1 (controlled)>
<State 2 (controlled)>
Place={p1,p2,p3,p4}
Transition={t1,t2,t3,t4}
tokens(p1)=0
tokens(p2)=1
tokens(p3)=2
tokens(p4)=3
</State 2 (controlled)>
<State 3 (controlled)>
Place={p1,p2,p3,p4}
Transition={t1,t2,t3,t4}
tokens(p1)=1
tokens(p2)=1
tokens(p3)=0
tokens(p4)=0
</State 3 (controlled)>
<State 4 (controlled)>
Place={p1,p2,p3,p4}
Transition={t1,t2,t3,t4}
tokens(p1)=0
tokens(p2)=2
tokens(p3)=3
tokens(p4)=0
</State 4 (controlled)>
...
```

Figure 4.1: Simulation of the Petri net AsmetaL model shown in Code 2.1

AsmetaS [80] is the ASMs simulator of the ASMETA framework, for the simulation of ASMs models written in AsmetaL. As key features for model validation, AsmetaS supports:

1. *invariant checking* to check whether invariants expressed over the ASM model under execution are satisfied or not. Invariants are checked after each step and, if a violation is detected, the simulation is interrupted raising an error. Invariants can be added to the model to check that some expected properties of the internal behaviour of the machine hold: in this case, a violation of an invariant must lead to a review of the model to fix the wrong behaviour. Moreover, invariants could be added to check that the ASM is accessed in the *right way*, namely that the values provided by the environment for the monitored functions respect some input contracts. Indeed, there could be constraints on the values that a monitored function can assume in certain states, and these constraints can be easily encoded as invariants[1].

2. *consistent updates checking* for revealing inconsistent updates (see Def. 2.6). The presence of an inconsistent update is a clear sign of a weakness of the model. It reveals that, under some conditions, the machine takes two decisions that conflict each other, since it wants to update the same location to two different values.

3. *interactive simulation* when required inputs (i.e., values for monitored functions) are interactively provided during simulation by the user. This is the standard way of simulation in which the user interacts with the model providing the correct inputs and judging the correctness of the observed behaviour.

4. *random simulation* when the values for the monitored functions are given randomly by the simulator. This kind of simulation is particular useful for exercising the model several times with minimal effort from the user. The aim of this activity is mainly to look for states in which an invariant is violated or an inconsistent update happens.

Moreover the simulator permits to execute a *limited simulation*, specifying the number of steps to execute, or an *unlimited simulation*, requiring that the simulation is stopped only when the update set is empty.

A simulation with AsmetaS produces as output a sequence of states where, in each state, only the locations updated/read up to that moment are shown.

Fig. 4.1 shows the first four steps of simulation of the AsmetaL model shown in Code 2.1. Since the model does not contain monitored functions, no user intervention is required. Since the model is nondeterministic, a different run of the simulator could obtain a different trace. In

[1]An alternative could be to encode the constraints directly in the model transition rules, but this would unnecessarily complicate the model.

```
asm sluiceGateMotorCtl
import StandardLibrary
signature:
    dynamic controlled phase: Phase
    dynamic monitored passed: TimePeriod -> Boolean
    dynamic monitored event: Position -> Boolean
     ...
definitions:
     ...
    main rule r_Main =
        par
            if(phase=FULLYCLOSED and passed(CLOSED_PERIOD)) then ...
            endif
            if(phase=OPENING and event(TOP)) then ...
            endif
            if(phase=FULLYOPEN and passed(OPEN_PERIOD)) then ...
            endif
            if(phase=CLOSING and event(BOTTOM)) then ...
            endif
        endpar
     ...
```

Code 4.1: Fragmet of the AsmetaL model for the Sluice gate problem

State 2, for example, the transition $t3$ of the modeled Petri net (see Fig. 2.4) has been fired, but also transition $t2$ could have been fired. For validating the model, an invariant has been added to check that all the places can not assume a negative number of tokens: in this case, a violation of the invariant would mean that the model is not correct. Note that, in the simulation trace, also the elements of the abstract domains *Place* and *Transition* are reported. The content of an abstract domain is shown because it could be modified by an extend rule (see Section 2.1.2.2) which adds a fresh element to it.

Code 4.1 shows a fragment of the AsmetaL implementation of the ASM introduced in [31] that models a sluice gate[2]. In the specification, in order to abstractly model the time, the boolean monitored function *passed* indicates if two time periods have been passed: the monitored locations *passed(CLOSED_PERIOD)* and *passed(OPEN_PERIOD)* indicate if the period of time during which the gate must stay, respectively, closed and open is elapsed. The monitored locations *event(TOP)* and *event(BOTTOM)*, instead, indicate if the gate has reached, respectively, the top and the bottom position. Fig. 4.2 shows the simulation of the model. Note that, at each step, the simulator asks the user for the values of the monitored locations; since the simulator executes a lazy evaluation of the conditions, it asks the values only for those monitored locations that are necessary to compute the update set. The print of the state is divided between the *monitored* part and the *controlled* part.

4.2 Scenario-based validation of ASMs

Scenario-based validation can be seen as the automation of the simple validation that can be obtained with basic simulation. It consists in building *scenario*s where to specify the inputs for exercising the model under validation, and the expected model behaviour.

In the ASMETA framework, scenario-based validation is done using the *AsmetaV* tool [39]. It provides a language, *Avalla*, to

- specify the AsmetaL model under validation;
- describe execution scenarios as sequences of actions. The user can

[2]A sluice, with a rising and falling gate, is controlled by a computer. The gate must be kept open (in the *top* position) for ten minutes in every three hours and otherwise kept closed (in the *bottom* position).

```
Insert a boolean for passed(CLOSED_PERIOD):
true
<State 0 (monitored)>
passed(CLOSED_PERIOD)=true
</State 0 (monitored)>
<State 1 (controlled)>
dir=CLOCKWISE
motor=ON
phase=OPENING
</State 1 (controlled)>
Insert a boolean for event(TOP):
false
<State 1 (monitored)>
event(TOP)=false
</State 1 (monitored)>
<State 2 (controlled)>
dir=CLOCKWISE
motor=ON
phase=OPENING
</State 2 (controlled)>
Insert a boolean for event(TOP):
true
<State 2 (monitored)>
event(TOP)=true
</State 2 (monitored)>
<State 3 (controlled)>
dir=CLOCKWISE
motor=OFF
phase=FULLYOPEN
</State 3 (controlled)>
Insert a boolean for passed(OPEN_PERIOD):
...
```

Figure 4.2: Simulation of the Sluice gate AsmetaL model shown in Code 4.1

  - **set** the environment (i.e., the values of monitored/shared functions);
  - **check** the machine state, observing the values of its functions;
  - force the machine to make one **step**, or a sequence of steps until a condition becomes true (by the command **step until**);
  - ask for the **exec**ution of new transition rules not contained in the model. Using this feature, the user can deeply influence the simulation and modify the behaviour of the ASM under validation; this results particular useful when validating nondeterministic models, since it permits to override nondeterministic choices that, otherwise, make it difficult to exactly specify the expected behaviour.

- add invariants that must hold during the scenario execution. Note that these invariants distinguish from those added in the AsmetaL models, since scenario invariants are required to hold only during the simulation of the scenario, whereas model invariants must hold in any possible simulation.

Depending on the actions specified in the scenario, the external actor interacting with the model is identified as *user* or *observer*. A *user* has a *black box view* of the system, since she can only set the value of the monitored functions, force machine steps, and check only the output functions of the machine. The *observer*, instead, has a *gray box view* of the model, since she can also check the machine controlled state and require the execution of new transition rules.

Code 4.2 shows a scenario for the AsmetaL model of the sluice gate, shown in Code 4.1. For four times, it **set**s the value of a proper monitored location, makes a **step**, and **check**s that the controlled state of the machine has been updated correctly.

AsmetaV works as follows:

1. it reads an Avalla user scenario *scen* and the AsmetaL specification *spec* which the scenario refers to;

2. it builds a new AsmetaL specification *specForV* starting from *scen* and *spec*. The simulation of *specForV* corresponds to the execution specified in the scenario *scen* over the original specification *spec*.

3. it *executes* the scenario by invoking the simulator AsmetaS over *specForV*. The scenario simulation is similar to a traditional model simulation, but, in addition, it also contains the results of the behaviour checks specified in the scenario with the command **check**.

```
scenario sluiceGate1

load sluiceGateMotorCtl.asm

set passed(closedPeriod) := true;
step
check phase = OPENING and motor = ON and dir = CLOCKWISE;

set event(top) := true;
step
check phase = FULLYOPEN and motor = OFF and dir = CLOCKWISE;

set passed(openPeriod) := true;
step
check phase = CLOSING and motor = ON and dir = ANTICLOCKWISE;

set event(bottom) := true;
step
check phase = FULLYCLOSED and motor = OFF and dir = ANTICLOCKWISE;
```

Code 4.2: Scenario for the Sluice gate AsmetaL model shown in Code 4.1

During the scenario execution, AsmetaV also collects data about the coverage of the transition rules of the original specification *spec*. This permits to check which transition rules have been exercised and which rules, instead, would require to be executed by different scenarios.

Chapter 5

Model review

Model review, also called *model walk-through* or *model inspection*, is a validation technique in which models are critically examined to determine if they, not only fulfills the intended requirements, but also are of sufficient quality to be easy to develop, maintain, and enhance. This process should, therefore, assure a certain degree of quality. The assurance of quality, namely ensuring readability and avoiding error-prone constructs, is one of the most essential aspects in the development of safety-critical reactive systems, since the failure of such systems – often attributable to modeling and, therefore, coding flaws – can cause loss of property or even human life [151]. When model reviews are performed properly, they can have a big payoff because they allow defects to be detected early in the system development, reducing the cost of fixing them.

Model review has been inspired by the review process executed during the software development. In Section 5.1 we introduce the technique of *program review*, while in Section 5.2 we describe some model review techniques.

In Section 5.3 we propose a general approach for doing model review of state-based formal notations. Then, the general approach is instantiated for NuSMV models and for the ASMs in Chapters 6 and 7.

5.1 Program review process

In the '70s at the IBM, Fagan proposed a technique for reviewing programs during the software development cycle (this process is also known as *Fagan inspection* [67]). In this process the code, but also any document that has been produced during the software development process (as designs, test plans, users manuals, etc.), are checked to see if they assure some *quality standards*.

Program review has demonstrated to be a very effective technique: in [67] it is reported that more than 60% of errors in a program has been discovered using a program inspection technique.

Although there are different variations of this process, it is usually divided into three phases [157]:

1. **Pre-meeting activities** In this phase the review team is composed by taking a group of qualified people, often both technical staff and project stakeholders, who usually are not involved in the development phase. Then the documents to be reviewed are distributed among the components of the team. Then, after a meeting in which the team discusses about the software under review, each reviewer individually inspects the documents to find faults, omissions, departures from standards, etc., and records *comment*s about them.

2. **Review meeting** In this phase the components of the review team meet together with a developer of the program and discuss about the comments recorded in the previous phase. The discussion about each comment must clarify if a real fault/omission in the code has been discovered and, in case, decide to fix it.

3. **Post-meeting activities** In this phase the problems in the code discovered in the previous phases are addressed. This can require fixing the code, refactoring it, or modifying/rewriting related documents. After this phase, a new review process could be made to check that all the problems revealed by the review process have been solved.

When the reviewers individually check the code (code review), they usually look for common programming errors [157] that can frequently occur and that are independent of the particular program under review (i.e., no knowledge of the program is needed to find them): e.g., an array is accessed outside its bounds, an input variable is not used, the types of the formal and the actual parameter do not match, etc.. Obviously, the kind of errors that reviewers must look for depends on the programming language used, since each language has its own typical errors (e.g., dangling pointers in C) and, instead, ensures that some other errors can not occur thanks to the way it has been designed (e.g., buffer overflows can not occur in Java).

For different languages, there are several tools that automatically look for common errors, as, for example, FindBugs[1], PMD[2] and Checkstyle[3] for Java, or Splint[4] for C. These tools look for erroneous code but also for *stylistic conventions* violations that may indicate a possible problem. For example, the pattern *Unwritten field* of FindBugs signals if a field has never been written and always returns its default value: the violation of this pattern could show that the field is not necessary or that we forgot to update it somewhere.

Using these tools in code review permits the reviewers to avoid looking for these trivial errors, and concentrate their attention on more subtle errors/stylistic violation that are more difficult to find with an automatic tool.

5.2 Model review techniques

As it has been done for code, also for models, depending on the notation used, it is possible to identify some common errors that can be easily identified. For an UML state machine, for example, an error is the presence of *miracle states* [151], i.e., non-initial states not having incoming transitions: Fig. 5.1 shows an UML state machine where B is a miracle state.

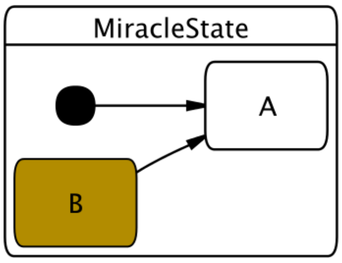


Figure 5.1: UML state machine with a *miracle state*

A weak aspect of a model review process is that, usually, it must be executed by hand. This requires a great effort that might be tremendously reduced if performed in an automatic way by systematically checking specifications for known vulnerabilities or defects. In a report about the certification of the Darlington nuclear plant, Parnas observed that "reviewers spent too much of their time and energy checking for simple, application-independent properties" which distracted them from the more difficult, safety-relevant issues" [144]. Tools that automatically perform such checks can save reviewers considerable time and effort, liberating them to do more creative work.

So, given a formal notation, the problems that arise are *what* properties must be checked over the model and *how* to automatically check them. In other words, it is necessary to identify classes of faults and defects to check, and to establish a process by which to detect such deficiencies in the underlying model. If these faults are expressed in terms of formal statements, these can be assumed as a sort of "measure" of the *model quality assurance.* A tool is also necessary to make the process automatic. It would work as *model advisor* to check a model for conditions

[1]`http://findbugs.sourceforge.net/`
[2]`http://pmd.sourceforge.net/`
[3]`http://checkstyle.sourceforge.net/`
[4]`http://www.splint.org/`

and configuration settings that can result in inaccurate or under-/over-specified behaviour of the system that the model represents.

Typical automatic reviews of formal specifications include simple syntax checking and type checking. This kind of analysis is performed by simple algorithms which are able to immediately detect faults like wrong use of types, misspelled variables, and so on. Some complex type systems may require proving of real theorems, like the non-emptiness of PVS types [142].

In the following sections, we will make a non-exhaustive overview of some model inspection techniques for different notations, that can fall into our definition of model review.

5.2.1 Software Cost Reduction

A model review technique has been developed for the Software Cost Reduction (SCR) method [94]. SCR is a requirements specification method that uses a tabular notation to define mathematical functions. There are different tables: *condition*, *event*, and *mode transition* tables. Each table describes a *variable* or a *mode* as a function of modes and/or events and/or conditions. For example, a condition table describes a controlled variable as a function of a mode and a condition, where a controlled variable is an environmental quantity that the system controls, a mode represents the state of an environment entity that influences the system, and a condition is a predicated defined on one or more system entities. Table 5.1 shows the condition table of the controlled variable `SafetyInjection` (taken from [94]), whose value is a function of `Pressure` and `Overridden`.

Mode of `Pressure`	**Conditions on `Overridden`**	
High, *Permitted*	*true*	*false*
TooLow	`Overridden`	¬ `Overridden`
`SafetyInjection`	*Off*	*On*

Table 5.1: Condition table of the controlled variable `SafetyInjection`

Reading the table is easy. If `Pressure` is *High* or *Permitted* (independently of the value of `Overridden`), or if `Pressure` is *TooLow* and `Overridden` is *true*, then `SafetyInjection` is *Off*; if `Pressure` is *TooLow*, and `Overridden` is *false*, then `SafetyInjection` is *On*. The entry *false* in the first row means that `SafetyInjection` can not be *On* when `Pressure` is *High* or *Permitted*.

The authors defined a *formal requirements model* specifying the properties that any SCR specification must satisfy, and developed a tool, the *consistency checker*, for checking these properties. They identified eight categories of properties: *Proper Syntax*, *Type Correctnesses*, *Completeness of Variable and Mode Class Definitions*, *Initial Values*, *Reachability*, *Disjointness*, *Coverage* and *Lack of Circularity*.

Let's give the definitions of three of these properties. *Type Correctnesses* requires that the types of the variables are satisfied in the tables. *Disjointness* requires that, in a given state, each controlled variable is uniquely defined, i.e., its conditions on each row are disjoint; the aim of this property is to check that the specifications are deterministic. *Coverage* requires that each variable described by a condition table is always defined. Table 5.2 is a modified version of Table 5.1 in which the three previous properties are violated.

Type Correctnesses is violated by the values assigned to variable `SafetyInjection`, since the variable domain is {*Off*, *On*} and not {*false*, *true*}.

To check the *disjointness* property, we must check that the conditions on each row (e.g., $cond_1$ and $cond_2$) are disjoint, that is that $\neg(cond_1 \wedge cond_2)$ is a tautology. But, we see that conditions on the second row of Table 5.2 do not satisfy the *disjointness* property since $\neg($`Overridden` $\wedge$ `Overridden`$)$ is not a tautology.

To check the *coverage* property, we must check that, in each state, at least a condition on each row is satisfied so that a value can be determined; for example, given conditions $cond_1$ and

Mode of `Pressure`	**Conditions on** `Overridden`	
High, *Permitted*	*true*	*false*
TooLow	`Overridden`	`Overridden`
`SafetyInjection`	*false*	*true*

Table 5.2: Condition table of the controlled variable `SafetyInjection` that violates properties *Type Correctnesses*, *Disjointness* and *Coverage*

$cond_2$, we must require that $cond_1 \vee cond_2$ is a tautology. But, we see that conditions on the second row of Table 5.2 do not satisfy the *coverage* property since `Overridden` $\vee$ `Overridden` is not a tautology.

In the SCR toolset, the checking of the *disjointness* and *coverage* properties are executed through the Consistency Checker tool that implements a tableau-based decision procedure. Since the consistency checker can not handle predicates containing complex numerical constraints, other tools have been experimented.

Salsa [27] is an invariant checker for specifications written in the SCR Abstract Language (SAL): it can check the validity of formulae on Boolean, enumerated and integer variables restricted to Presburger arithmetic. For specifications containing numerical variables, that were not supported by the consistency checker tool, Salsa could perform the analysis [27].

In [37], instead, the infinite state model checker Action Language Verifier (ALV) has been used. In this case, the properties to verify are CTL formulae that predicate over the values assumed by the variables. For example, for checking the *disjointness* property, for each dependent variable d, the following CTL property must be checked

$$\mathbf{AG}\left(\mathbf{EX}\left(d = v_d \wedge \bigwedge_{m \in M} m = v_m\right) \Rightarrow \mathbf{AX}\left(\bigwedge_{m \in M} m = v_m \Rightarrow d = v_d\right)\right)$$

where M is the set of monitored variables, and v_d and v_m are type-correct values for variables d and m. In SCR, the values of dependent variables in a state depend on the values of the monitored variables. The *disjointness* property requires that the system is deterministic and so that, for any state, the value of any dependent variable in the next state is uniquely determined by the values of the monitored variables in the next state. So, given a current state S, there can not be a next state $\hat{S}'$ in which the monitored variables take values $\{v_m^1, \dots, v_m^n\}$ and the dependent variable takes value $\hat{v}_d$, and another next state $\tilde{S}'$ in which the monitored variables take the same values $\{v_m^1, \dots, v_m^n\}$ but the dependent variable takes value $\tilde{v}_d$ with $\hat{v}_d \neq \tilde{v}_d$.

5.2.2 SCR-style specifications

In [114] a method to automatically verify the consistency of software requirements specifications (SRS), written in SCR-style, is described. SCR-style specifications are similar to SCR specification: the difference lies in how primitive functions are described.

The authors noticed that normal inspections of SCR-style specifications – following the process described by Fagan in [67] – became ineffective when used to verify structural properties of *large, complex, and evolving requirements.* Structural properties are application independent properties that requires that the definition and use of variables, functions and functions groups are consistent. They observed that reviewers find very tedious checking for this properties, since it is a process very repetitive and less stimulating.

The idea of their work was to use the Prototype Verification System (PVS) [142] to automatize the verification of structural properties on SCR-style specifications. The verification process works as follows:

1. An SCR-style specification is translated into a PVS specification using an algorithm described in the paper;

2. for each property, a PVS theorem is built;
3. the user selects a property to prove and PVS automatically executes the proof without the need of user intervention.

For each property, the authors defined a PVS theorem that must be added to the PVS specification (step 2 of the process) obtained from the translation of the SCR-style specification. All the theorems can be proved automatically, so that the user does not have to guide the proof and no knowledge of PVS is required. Since for one property it was not possible to build a theorem that could be automatically verified, such property has been encoded as a CTL property and its verification is carried out using the model checking capabilities of PVS.

The authors identified the following properties:

P1 Each external input should be used in at least one function.

P2 Each external output should be generated by a function.

P3 All internal data flows must be generated from a source function and consumed by target functions.

P4 All data flows declared in higher levels of a FOD are consistent with the ones defined in lower levels and vice versa.

P5 The data flows of one function are consistent with the input–output relation of the function definition body written in a structured decision table.

P6 No circular dependencies exist among data flows.

Properties P1-5 are encoded as PVS theorem, whereas property P6 is encoded as a CTL formula.

Let's see, as an example, property P1. It requires that each monitored function is actually useful in the definition of at least a function. The PVS theorem that must be proved is:

```
monitor_check: THEOREM
    FORALL (m_var : monitor_type): EXISTS (f_var : function_type):
    member((m_var, f_var), dependency_set)
```

It requires that, for each monitored function `m_var`, it exists a function `f_var`, such that in the set `dependency_set`[5] it exists a tuple `(m_var, f_var)`: this means that `f_var` depends on `m_var`.

5.2.3 Statecharts

Statecharts [90] are *an extension of state machines and state diagrams for the specification and design of complex discrete event-systems.* They provide the notions of hierarchy, orthogonality, compound events, and a broadcast mechanism for the communication of concurrent components. The UML state machines [164] are an object-based variant of Harel statecharts.

In [151], the authors present a set of rules that seek to avoid common types of errors by ruling out certain modeling constructs for UML state machines or Statecharts.

The authors state that the first rules that must be respected are the UML *well-formedness rules* [164]. These rules are expressed as OCL constraints over UML models; the satisfaction of these constraints assures the syntactical correctness, which is a prerequisite for executing more complex checks. An example of *well-formedness rule* is the rule *CompositeState-1* that states that *a composite state can have at most one initial vertex.*

The authors then reviewed different style guides proposed for statecharts and their dialects. They devised two categories of rules:

[5]The `dependency_set` describes the dependency relations among the primitive functions.

- *Syntactical Robustness Rules* identify syntactical constructions that, although syntactically correct according to the *well-formedness rules*, should be avoided because they could produce misleading models;
- *Semantic Robustness Rules* try to detect incorrect model behaviours, e.g., race conditions.

Syntactical Robustness Rules The syntactical robustness rules derived from the literature are *MiracleStates*, *IsoatedStates*, *EqualNames*, *InitialState*, *OrStateCount*, *RegionStateCount* and *DefaultFromJunction*. In addition to these rules, the authors also identified the *TransitionLabels*, *InterlevelTransitions* and *Connectivity* rules.

In the introduction of Section 5.2 we have already seen the *MiracleState* rule that requires that, except the initial state, all the states must have an incoming transition. Fig. 5.1 shows a violation of such property. An extension of the *MiracleState* rule is the *Connectivity* rule that requires that each state must be reachable from the initial state. Fig. 5.2 shows a machine that violates the *Connectivity* rule, although it does not violate the *MiracleState* rule.

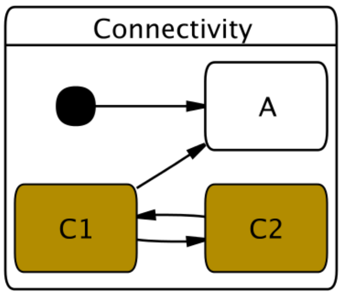


Figure 5.2: UML state machine with a violation of the *Connectivity* rule

The checking of syntactical robustness rules, as well as of well-formedness rules, is done using OCL constraints. OCL, indeed, is powerful enough to reason about the syntactical structure of a machine.

Semantic Robustness Rules The semantic robustness rules identified are *Transition Overlap*, *Dwelling* and *Race Conditions*. The *Transition Overlap* rule, for example, requires that all transitions outgoing from a state should have semantically disjoint predicates. Fig. 5.3 shows a state whose exiting transition should be checked for disjointness.

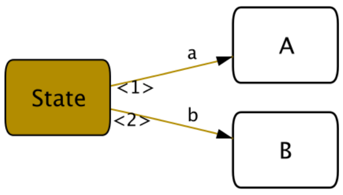


Figure 5.3: UML state machine with a possible violation of the *Transition Overlap* rule

In the proposed approach, the authors, for checking the semantic robustness rules, have to use an SMT solver, since, for predicating about the behaviour of a machine, OCL is not enough. For example, for checking the *Transition Overlap* rule for the state in Fig. 5.3, the satisfiability of the following formula is checked

$$(e_a \wedge c_a) \wedge (e_b \wedge c_b)$$

```
MODULE main
VAR
    hour: 0..23;
    hour12: 1..12;
    amPm: {AM, PM};
ASSIGN
    init(hour) := 0;
    next(hour) := (hour + 1) mod 24;
    hour12 :=
        case
            hour in {0, 12}: 12;
            !(hour in {0, 12}): hour mod 12;
        esac;
    amPm :=
        case
            hour < 12: AM;
            hour > 11: PM;
        esac;

CTLSPEC AG(hour > 11 -> amPm = PM);
```

Code 5.1: NuSMV model of a clock with a CTL property non-vacuously satisfied

where e_a and c_a are the event and the condition of transition a, and e_b and c_b of transition b. If the formula is satisfiable, it means that transitions a and b are not disjoint.

5.2.4 Vacuity detection

Model checkers check systems correctness with respect to some desired behaviours specified as temporal properties (see Section 9.1). Most model checkers tools, upon a property violation, provide a *counterexample*, in the form of an execution trace of the model, showing why the property does not hold. These counterexamples are very useful because permit to the modelers to discover errors in their models. When the property is satisfied by the model, instead, most of the tools simply say that the property holds, without providing any *witness* of why the property holds. But, if the property holds, it does not necessarily mean that the model of the system is correct: the property could be true for the *wrong* reason.

Code 5.1 shows the NuSMV model of a clock that memorizes only the hours: the variable *hour* memorizes the hour in the 24-hour format, whereas variables *hour12* and *amPm* provide the hour in the 12-hour format. Variable *hour* is initialised to zero and it is incremented of a unit (modulo 24) in each transition from a state to the next one. The values of *hour12* and *amPm* are defined based on the value of *hour*. A CTL property checks that, in each state, if *hour* is greater than 11, then *amPm* is *PM*.

Code 5.2 shows a modified version of the model in Code 5.1 in which the variable *hour* is always 0.

In both models the CTL property is satisfied. However, in the model in Code 5.2 the property is true because subformula $hour > 11$ (antecedent of the implication) is always false: the property is *vacuously satisfied*.

In [24] the authors say that, according to their experience, *typically 20% of formulae pass vacuously during the first formal verification runs of a new hardware design, and that vacuous passes always point to a real problem in either the design or its specification or environment.* So, discovering properties that are vacuously satisfied is extremely important. Since vacuity detection is independent of a particular model, we can classify it as a model review technique.

In [24], a technique has been proposed for checking the vacuity of formulae in w-ACTL, a subset of CTL without the operators existentially quantified, in which the $\neg$ operator modifies only atomic propositions, and in which for all the binary operators at least one of the operands is a propositional formula.

```
MODULE main
VAR
    hour: 0..23;
    hour12: 1..12;
    amPm: {AM, PM};

ASSIGN
    hour := 0;
    hour12 :=
        case
            hour in {0, 12}: 12;
            !(hour in {0, 12}): hour mod 12;
        esac;
    amPm :=
        case
            hour < 12: AM;
            hour > 11: PM;
        esac;

CTLSPEC AG(hour > 11 −> amPm = PM);
```

Code 5.2: NuSMV model of a clock with a CTL property vacuously satisfied

The technique has been extended in [118] for checking the vacuity of CTL* formulae [16]. CTL* is an extension of CTL and it subsumes both CTL and LTL. So, techniques for checking the vacuity of CTL* formulae, can be used for checking the vacuity of CTL and LTL formulae as well. We report the definition of vacuity satisfaction as presented in [118].

Definition 5.1. *A system M satisfies a formula φ vacuously iff $M \models \varphi$ and there is a subformula ψ of φ that does not affect φ in M.*

Intuitively, a subformula ψ does not affect φ in M if the truth value of φ is independent of the truth value of ψ, i.e.,

$$M \models \varphi[\psi \leftarrow false] \quad \text{iff} \quad M \models \varphi[\psi \leftarrow true]$$

where $\varphi[\psi \leftarrow \rho]$ is the formula φ where the subformula ψ is replaced by the formula ρ.

Actually, the algorithm for checking the vacuity of a formula is simpler. Indeed, when checking for vacuity, it is already known that $M \models \varphi$ and so it is not necessary to consider both replacements of values *false* and *true*. The algorithm uses the polarity of subformulae: the polarity of a subformula ψ is positive, if it is nested in an even number of negations in φ, otherwise it is negative. The algorithm works as follows. For each subformula ψ in φ:

1. the formula φ' is built based on the polarity of ψ. If ψ is positive, ψ is replaced by *false* in φ, i.e., $\varphi' = \varphi[\psi \leftarrow false]$, otherwise, if ψ is negative, ψ is replaced by *true* in φ, i.e., $\varphi' = \varphi[\psi \leftarrow true]$;

2. the formula φ' is model checked; if $M \models \varphi'$, it is said that φ is vacuously satisfied for the subformula ψ.

For example, let's apply the algorithm to the CTL formula

$$\mathbf{AG}(hour > 11 \rightarrow amPm = PM)$$

of the NuSMV model in Code 5.1 (let's call it M), and also of its modified version in Code 5.2 (let's call it M').

Applying the replacement based on the polarity of the subformulae $hour > 11$ and $amPm = PM$, generates the two following formulae[6]

$$\alpha = \mathbf{AG}(\mathit{TRUE} \rightarrow amPm = PM)$$
$$\beta = \mathbf{AG}(hour > 11 \rightarrow \mathit{FALSE})$$

The result of model checking M against α and β is

$$M \not\models \alpha \qquad M \not\models \beta$$

This means that the CTL property is non-vacuously satisfied in M. Instead, the result of model checking M' against α and β is

$$M' \not\models \alpha \qquad M' \models \beta$$

This means that the CTL property is vacuously satisfied in M. The verification of formula β shows that the value of the original formula does not depend on the value of the subformula $amPm = PM$: indeed, since the antecedent of the implication is always *false*, the implication is always *true*, nevertheless the value of the consequent.

5.3 Proposed model review approach

We here propose a general model review approach for state-based formal methods that has been concretely implemented for the NuSMV notation and for the ASMs, as described in Chapters 6 and 7. The choice of defining model review processes for NuSMV and ASMs is due to the fact that both are endowed with tools that permits to easily automatize the proposed approach, and provide a wide range of models to analyse.

Let's suppose to have a state-based formal method F for which we want to develop a model review technique. The general definition of a state-based formal method has been given in Def. 1.1 as $F = \langle S, S^0, T\rangle$, being S the set of states, S^0 the set of initial states, and T the transition relation. $\mathcal{R}(s)$ is the set of all the states reachable from s. Let M be a model of F.

In order to develop a model review technique for F, first of all we must identify defects, vulnerabilities, and deviations from the notation style guide that are usually introduced by a developer when modeling with F. Then these faults must be expressed as the violations of formal properties. These properties refer to model attributes and characteristics that should hold in any model, independently from the particular model to analyse. For this reason we call them *meta-properties* (we identify them as $\mathrm{MP_F}1, \ldots, \mathrm{MP_F}n$). They should be true in order for a model of F to have the required quality attributes. Therefore, they can be assumed as measures of model quality assurance. The violation of a meta-property always means that a quality attribute is not met and may indicate a potential/actual fault in the model. The severity of such violation depends on the meta-property, each of which measures the degree of model adequacy to the guidelines of the formal method modeling style.

Although the properties defined for a notation are different from those defined for another notation, we have devised three categories of model quality attributes that are general enough to cover different state-based formal methods:

- **Consistency** requires that there are no model statements that conflict with each other. Violations of consistency meta-properties usually identify real faults of the model. In NuSMV, for example, a consistency meta-property could require that there is no specified temporal property false: if the meta-property is violated, it means that the model contradicts the temporal property. In ASMs, instead, a consistency meta-property could require that a location is never simultaneously updated to two different values: if violated, it means that there are two update rules that contradict each other.

[6]Note that the antecedent of an implication is considered to be under negation.

- **Completeness** requires that every system behaviour is explicitly modeled. Violations of completeness meta-properties seldom identify real faults of the model: most of the times they identify parts of the model that could be difficult to understand by an external reader, because the behaviour of the machine is left unspecified (usually because the state of the machine does not have to change). These meta-properties force the modeler to explicitly describe the behaviour of the model in every state and under each condition. For example, both in NuSMV and in ASMs, a completeness meta-property could require that no *conditional* branch (a branch of a case expression in NuSMV, or the else branch of a conditional rule in ASMs) is left unspecified, although permitted by the notation.

- **Minimality** requires that the model does not contain elements defined or declared in the model but never used. These defects are also known as *over-specification.* Violations of minimality meta-properties usually identify unnecessary elements of the model, or they signal that the model is not complete. For example, both in NuSMV and in ASMs, a minimality meta-property could require that each variable is read at least once.

In order to describe the meta-properties, we define four logical operators that permits to specify *when* (i.e., in which states) a property must hold. The operator *Always* permits to capture properties that must be true in every state, and the operator *Sometime* properties that must be eventually true in at least one state. Moreover, the operator *InitiallyA* is used to describe properties that must be true in every initial state, and *InitiallyS* properties that must be true in at least one initial state. Their formal definition is as follows:

$$M \models \mathit{Always}(\varphi) \quad = \quad \forall s_0 \in S^0,\ \forall s \in \mathcal{R}(s_0) : \varphi(s) \tag{5.1}$$
$$M \models \mathit{Sometime}(\varphi) \quad = \quad \exists s_0 \in S^0,\ \exists s \in \mathcal{R}(s_0) : \varphi(s) \tag{5.2}$$
$$M \models \mathit{InitiallyA}(\varphi) \quad = \quad \forall s_0 \in S^0 : \varphi(s_0) \tag{5.3}$$
$$M \models \mathit{InitiallyS}(\varphi) \quad = \quad \exists s_0 \in S^0 : \varphi(s_0) \tag{5.4}$$

where φ is a predicate over a state of M.

How to execute the verification of these properties depends on the state-based formal method for which we are developing the model review technique. In the model reviews for NuSMV and ASMs, these properties are translated into CTL formulae and then model checked.

5.3.1 Evaluation of the approach in terms of fault detection capability

The last step of the development of a model review technique should be an assessment of its ability of detecting real faults. Indeed, the meta-properties introduced could target real faults of a model or only ”stylistic” defects (violations of style-guide conventions) that does not affect the behaviour of the model.

In Chapter 8 we propose a methodology for assessing the fault detection capability of a model review technique, based on *mutation analysis*, using an approach similar to mutation testing [106]. Our approach has been applied to the NuSMV model advisor, but it could be applied to any other model review technique.

Chapter 6

Model review of NuSMV models

We now tackle the problem of automatically reviewing NuSMV formal specifications (Section 3.2).

We here refer to the NuSMV formal specification method which is endowed with a simulator and a model checker that make possible to handle and to automatize our approach. Moreover, there exists a wide repository of NuSMV specification case-studies available for testing the model advisor we propose.

We develop a model advisor for NuSMV programs which helps the developers to assure given model qualities. We first detect a family of vulnerabilities and defects a developer can introduce during the modeling activity using NuSMV and we define appropriate meta-properties to capture them.

We have identified 10 meta-properties which use all the logical operators defined in Section 5.3, i.e., *Always*, *Sometime*, *InitiallyA* and *InitiallyS*. In order to verify these meta-properties, the logical operators are mapped to temporal logic formulae and the NuSMV model checking facilities are exploited to check for meta-property violations.

We only consider NuSMV models containing operational commands in this work. There are known techniques [41] for converting more general TRANS-based specifications into our operational form, so this is not a fundamental limitation.

Section 6.1 defines a function, later used in the meta-properties definition, that statically computes the assignment condition under which a model variable is updated upon state changing. Meta-properties able to guarantee certain quality attributes of a specification are introduced in Section 6.2. In Section 6.3, we describe how it is possible to automatize our model review process by exploiting the use of NuSMV itself as a model checker to check for possible violations of meta-properties. The general architecture of our model advisor is described in Section 6.4. As a proof of concept, in Section 6.5 we report the results of applying our model advisor to a certain number of benchmark models of various degree of complexity: some taken from the NuSMV source distribution, others found on the Internet, others obtained by translating ASM models of real case studies to NuSMV models. Considerations about the fault detection capability of the approach are reported in Section 6.6.

6.1 Assignment Condition

As stated in Section 3.2, there exist different ways to assign values to NuSMV variables. Formally, an assignment is a pair $\langle$*identifier, expr*$\rangle$ where *identifier* is a variable identifier and *expr* is a *simple* or *next*-expression which provides the variable value.

We here present a method to compute, for each assignment $\langle$*identifier, expr*$\rangle$ defined in the specification under review, the list of conditions under which the assignment is actually performed, and the condition-free expressions, which will determine the value assigned to the variable *identifier* when computed. For this purpose, we introduce a function *assignment condition*

$$AC : Assignment \rightarrow (Condition \times ValueExpr)^{+}$$

It is defined as $AC(\langle \mathit{identifier}, \mathit{expr}\rangle) = CV(\mathit{expr})$ in terms of the function

$$CV : \mathit{Expression} \rightarrow (\mathit{Condition} \times \mathit{ValueExpr})^{+}$$

which extracts the conditions from a generic expression by returning the list of pairs $\langle \mathit{cond}, \mathit{valExpr}\rangle$, where *cond* is the condition under which the expression takes the value given by the expression (without conditions) *valExpr*. CV is recursively defined as follows, depending on whether *expr* is defined in terms of a conditional operator or not.

Expression without conditions. In this case *expr* is a constant, an identifier, a logical/algebraic expression, etc. The function yields $CV(\mathit{expr}) = \langle \mathit{true}, \mathit{expr}\rangle$.

Expression with conditions. In this case *expr* is expressed in terms of a **case** or an **if-then-else** operator. Before defining the function CV in these cases, let us introduce two auxiliary functions.

Let $\oplus_{i=1}^{n}(L_i)$ be the concatenation of lists L_i, with $i = 1, \ldots, n$.

Let $L \doteq [\langle c_i, e_i\rangle]_{i=1}^{n}$ be a list of pairs $\langle c_i, e_i\rangle$ (with $i = 1, \ldots, n$), where c_i a boolean condition and e_i an expression. We define a function $\bigwedge_a(L) \doteq [\langle a \wedge c_i, e_i\rangle]_{i=1}^{n}$ returning the list of pairs obtained from the elements of L by making the conjunction between the boolean condition a with the condition c_i.

- If *expr* is an **if-then-else** expression, CV holds:

$$CV(c?e1 : e2) = \oplus\left(\bigwedge_{c} CV(e1), \bigwedge_{\neg c} CV(e2)\right)$$

- If *expr* is a **case** expression, the CV function yields:

$$CV\left(\begin{array}{l}\textbf{case}\\ \quad \mathit{left\_expr\_1} : \mathit{right\_expr\_1};\\ \quad \ldots\\ \quad \mathit{left\_expr\_n} : \mathit{right\_expr\_n};\\ \textbf{esac}\end{array}\right) = \oplus_{i=1}^{n}\left(\bigwedge_{\mathit{left\_expr\_i}} CV(\mathit{right\_expr\_i})\right)$$

Example Code 6.1 shows a fragment of a NuSMV model in which the next value of the variable x is defined using a case expression.

```
ASSIGN
    next(x) :=
        case
            a1:
                case
                    b1: 2;
                    b2: 3;
                esac;
            a2: c ? 5 : 6;
            TRUE: 7;
        esac;
```

Code 6.1: Definition of the next value of variable x

The following is the computation of the function AC on the next_expression.

$$AC(\langle x, \mathit{next\_expr}\rangle) = CV(\mathit{next\_expr}) =$$
$$[(a1 \wedge b1, 2), (a1 \wedge b2, 3), (a2 \wedge c, 5), (a2 \wedge \neg c, 6), (\mathit{true}, 7)]$$

6.2 Meta-properties

In this section we introduce ten meta-properties ($\mathrm{MP_N}$1-10) that should be proved in order to assure that a NuSMV specification has some quality attributes.

The three categories of model quality attributes introduced in Section 5.3 for state-based formal methods, can be adapted to the NuSMV notation in the following way:

- **Consistency** requires that there are no model statements (variable assignments, propriety specifications, behaviours, etc.) that conflict with each other. For instance, $\mathrm{MP_N}$9 requires that all the specified properties are true. Consistency of assignments to variables, one of the main goals of other model review techniques [7, 94], is guaranteed in NuSMV by the semantics of the language. However, one could require mutual exclusion of assignment conditions ($\mathrm{MP_N}$3).
- **Completeness** requires that the transition relation is explicitly described. This encourages the explicit assignment of variables ($\mathrm{MP_N}$7) and that at least one assignment condition, apart the default condition, is true ($\mathrm{MP_N}$4).
- **Minimality** guarantees that the specification does not contain elements – i.e., variables, assignments, type values, etc. – defined or declared in the model but never used. Minimality of the assignments requires that every assignment can be performed ($\mathrm{MP_N}$1, $\mathrm{MP_N}$2) and it is really useful ($\mathrm{MP_N}$5). Every value of a variable type should be necessary ($\mathrm{MP_N}$6) and every variable used ($\mathrm{MP_N}$7 and $\mathrm{MP_N}$8). Minimality of properties requires that property specifications are not vacuously satisfied ($\mathrm{MP_N}$10).

6.2.1 Meta-property definition

In the following we present the meta-properties we have introduced for automatic review of NuSMV models.

Most of the meta-properties are expressed in terms of the assignment condition function. For notational convenience, given an assignment $\alpha = \langle id, expr\rangle$, we denote by $AC_{\alpha,i}$ the condition $cond_i$ of the i-th element $\langle cond_i, val_i\rangle$ of the list $AC(\alpha)$. Moreover, we need to distinguish between assignments α regarding initial values, called α_{init} assignments, and non-initial assignments. For the sake of brevity, all the following meta-properties containing $AC_{\alpha,i}$ (resp. $AC_{\alpha_{init},i}$) are universally quantified over assignment α (resp. α_{init}) and condition index i.

All the logical operators defined in Section 5.3, i.e., *Always*, *Sometime*, *InitiallyA* and *InitiallyS* are used in the meta-properties.

$\mathrm{MP_N}$1 Every assignment condition can be true

We would like that every condition, under which a variable is assigned a value, can be eventually true, i.e., the model does not contain conditions which are always false. We have to distinguish between initial and non-initial assignments.

This meta-property requires that every condition can be true in at least one initial state (formula 6.1), and every non-initial condition is eventually true (formula 6.2).

$$\mathit{InitiallyS}(AC_{\alpha_{init},i}) \tag{6.1}$$
$$\mathit{Sometime}(AC_{\alpha,i}) \tag{6.2}$$

In Code 6.2, the condition $x = BB$ is never satisfied.

$\mathrm{MP_N}$2 Every assignment is eventually applied

Even if a condition φ can be true by $\mathrm{MP_N}$1, we would actually like φ to eventually be evaluated and not to be masked by other conditions preceding it in a case expression. In such a case, we may suspect that the conditions in a *case* expression are mistakenly listed. $\mathrm{MP_N}$2 is guaranteed by proving formula 6.3 for initial conditions and formula 6.4 for non-initial conditions.

```
MODULE main
  VAR
    x: {AA, BB, CC};
  ASSIGN
    init(x) := AA;
    next(x) :=
      case
        x = AA: CC;
        x = BB: AA;−−never satisfied
        x = CC: AA;
      esac;
```

Code 6.2: Violation of meta-property $\mathrm{MP_N}1$

```
MODULE main
  VAR
    x: 1..3;
  ASSIGN
    init(x) := 1;
    next(x) :=
      case
        x = 1: 2;
        x > 1: {1, 3};
        x = 3: 1;−− never applied
      esac;
```

Code 6.3: Violation of meta-property $\mathrm{MP_N}2$

$$\mathit{InitiallyS}\left(AC_{\alpha_{init},i} \wedge \bigwedge_{j=1}^{i-1} \neg AC_{\alpha_{init},j}\right) \tag{6.3}$$

$$\mathit{Sometime}\left(AC_{\alpha,i} \wedge \bigwedge_{j=1}^{i-1} \neg AC_{\alpha,j}\right) \tag{6.4}$$

In Code 6.3, the condition $x = 3$ is eventually satisfied ($\mathrm{MP_N}1$) but the corresponding assignment is never applied because the condition is masked by the previous condition $x > 1$.

$\mathrm{MP_N}3$ The assignment conditions are mutually exclusive

This meta-property requires that every condition explicitly and precisely models the conditions under which the assignment is applied. This guarantees that, if the condition is true, it is applied and it is not masked by another condition which precedes it. $\mathrm{MP_N}3$ is guaranteed by proving formula 6.5 for initial conditions and formula 6.6 for non-initial conditions.

$$\forall j,\ 1 \leqslant j < i \quad \mathit{InitiallyA}(\neg(AC_{\alpha_{init},i} \wedge AC_{\alpha_{init},j})) \tag{6.5}$$

$$\forall j,\ 1 \leqslant j < i \quad \mathit{Always}(\neg(AC_{\alpha,i} \wedge AC_{\alpha,j})) \tag{6.6}$$

In Code 6.4, even if the model is correct, i.e., the value of *amPm* and *hour12* are correctly related to the value of *hour*, the two conditions of the assignment of the variable *amPm* are not mutually exclusive. To remove the violation we could, for example, change the second condition with the condition *hour* > 11.

```
MODULE main
  VAR
    hour: 0..23;
    hour12: 1..12;
    amPm: {AM, PM};
  ASSIGN
    init(hour) := 0;
    next(hour) := (hour + 1) mod 24;
    hour12 :=
      case --conditions mutually exclusive
        hour in {0, 12}: 12;
        !(hour in {0, 12}): hour mod 12;
      esac;
    amPm :=
      case --conditions not mutually exclusive
        hour < 12: AM;
        hour >= 11: PM;
      esac;
```

Code 6.4: Violation of meta-property $\mathrm{MP_N3}$

```
MODULE main
  VAR
    x: 2..4;
  ASSIGN
    init(x) := 2;
    next(x) :=
      case
        x = 2: 4;
        x = 4: 3;
        TRUE: 2; --default condition useful
      esac;
```

Code 6.5: Violation of meta-property $\mathrm{MP_N4}$

$\mathrm{MP_N4}$ For every assignment terminated by a default condition *true*, at least one assignment condition is true

We have already discussed in Section 3.2 that the conditions in a case expression must be *complete*. However, sometimes NuSMV forces the user to add a last default condition equal to *true*, even if the conditions in the case expression are already complete. This is due to fact that the completeness check executed by NuSMV is based on a static analysis of the code, that could give imprecise results in some cases. The following meta-property requires that all the conditions before the last default condition are already *complete*. If, for an initial assignment, $AC_{\alpha_{init},n} = true$, then formula 6.7 must be checked. If, for a non-initial assignment, $AC_{\alpha,n} = true$, then formula 6.8 must be checked.

$$InitiallyA(AC_{\alpha_{init},1} \vee \cdots \vee AC_{\alpha_{init},n-1}) \tag{6.7}$$

$$Always(AC_{\alpha,1} \vee \cdots \vee AC_{\alpha,n-1}) \tag{6.8}$$

This applies only when $AC_{\alpha,n} = true$ (or $AC_{\alpha_{init},n} = true$), because, otherwise, NuSMV already guarantees completeness. If this meta-property is verified, the default condition is useless because the previous conditions already cover every case.

In Code 6.5 the meta-property is violated, i.e., $Always(\mathrm{x} = 2 \vee \mathrm{x} = 4)$ is false, because, in the next expression of variable x, the default condition is useful since the previous conditions do not cover all the cases. The meta-property would be not violated if the next expression was

```
MODULE main
  VAR
    shuffle : boolean;
    x: {AA, BB};
  ASSIGN
    next(x) :=
      case
        ! shuffle  & x = AA: AA;--trivial
        ! shuffle  & x = BB: BB;--trivial
        shuffle : {AA, BB};
      esac;
```

Code 6.6: Violation of meta-property $\mathrm{MP_N5}$

rewritten, for example, in the following way:

```
next(x) :=
  case
    x = 2: 4;
    x = 4: 3;
    x = 3: 2;
  esac;
```

$\mathrm{MP_N5}$ No assignment is always trivial

We say that a *next* assignment $\langle var, expr\rangle$ is trivial if *var* is already equal to *expr*, even before the update is applied. This property requires that each assignment which is eventually performed, will not be always trivial, unless it is explicitly formalized by the assignment $\langle var, var\rangle$ which assigns to *var* its current value. The property

$$Sometime\left(AC_{\alpha,i} \wedge \bigwedge_{j=1}^{i-1} \neg AC_{\alpha,j}\right) \rightarrow Sometime\left(AC_{\alpha,i} \wedge \bigwedge_{j=1}^{i-1} \neg AC_{\alpha,j} \wedge var \neq expr\right) \quad (6.9)$$

states that, if eventually updated (see $\mathrm{MP_N2}$), the variable *var* will be updated to a new value at least in one state. The more simple property $Sometime(AC_{\alpha,i} \wedge var \neq expr)$ would be false if the assignment is never applied.

We borrowed the concept of trivial update from the ASMs [32].

In Code 6.6, in the next expression of variable x, the first two assignments are always trivial. The next expression could be rewritten in a more simple equivalent way:

```
next(x) :=
  case
    ! shuffle : x;
    shuffle : {AA, BB};
  esac;
```

$\mathrm{MP_N6}$ Every variable can take any value in its type

This meta-property requires that every variable takes all the values of its type. For each variable *var*, whose type values are $e_1, \ldots, e_n$, the property

$$Sometime(var = e_1) \wedge \ldots \wedge Sometime(var = e_n) \quad (6.10)$$

states that variable *var* takes all the values of its type. Since each $Sometime(var = e_i)$ is checked individually, we can know all the values never taken; these values, if they are really unnecessary, can be removed from the type definition.

In Code 6.7, variable x never takes value 2. Note that different variables can be defined over the same values, as x and y in the example. However, for NuSMV their types are distinct and

```
MODULE main
  VAR
    x: {1, 2, 3}; --it never takes value 2
    y: {1, 2, 3}; --it takes all the values
  ASSIGN
    init(x) := 1;
    next(x) := (x * 3) mod 4;
    y := {1, 2, 3};
```

Code 6.7: Violation of meta-property $\mathrm{MP_N}6$

```
MODULE main
  VAR
    x: boolean;
    xMU: boolean;--monitored variable used
    xMNU: 1..3;--monitored variable not used
    xMEA: boolean;--variable explicitly assigned
  ASSIGN
    x := !xMU;
    xMEA := {FALSE, TRUE};
```

Code 6.8: Violation of meta-property $\mathrm{MP_N}7$

so they can be considered individually: indeed, we can remove value 2 from the type of variable x without affecting the type of variable y that, instead, takes all its values.

$\mathrm{MP_N}7$ Every variable not explicitly assigned is used

In NuSMV there is no definition of monitored variables, i.e., variables that are updated by the environment. However, the variables that are not explicitly defined by an init/next/simple assignment, can be considered as monitored, since, at every step, they can take any value of their type. We claim that monitored variables should be *used* in other parts of the model, or that it should be made explicit that they can take any value (with a nondeterministic assignment over all the variable type). We say that a variable is *used* if it occurs in an assignment (in the right-hand side of an **ASSIGN** expression) or in a property (a **CLTSPEC/LTLSPEC** section).

The verification of this meta-property is statically performed by analysing the model without the use of the proving capabilities of the model checker.

In Code 6.8 *xMU* and *xMNU* are both monitored variables. *xMU* satisfies the meta-property because it is used in the simple assignment of variable x; *xMNU*, instead, violates the meta-property because is never read. The variable *xMEA*, instead, satisfies the meta-property because is explicitly assigned.

$\mathrm{MP_N}8$ Every independent variable is used

In NuSMV a variable x can be assigned in the next state to a value which depends only on x. In this case we say that the variable is *independent*, since it does not depend on other variables. Independent variables are generally used to model monitored variables which however have some constraints for their behaviour. These variables should be used in other parts of the model.

In Code 6.9 *xIU* and *xINU* are both independent variables. *xIU* satisfies the meta-property because is used in the assignment of variable x; *xINU*, instead, violates the meta-property because is never read.

$\mathrm{MP_N}9$ Every property is proved true

This meta-property simply requires that every property (CTL and LTL properties, and invariants) is proved true.

```
MODULE main
  VAR
    x: boolean;
    xIU: boolean;--independent variable used
    xINU: 0..4;--independent variable not used
  ASSIGN
    x := !xIU;
    init(xIU) := TRUE;
    next(xIU) := !xIU;
    init(xINU) := 0;
    next(xINU) := (xINU + 1) mod 5;
```

Code 6.9: Violation of meta-property $\mathrm{MP_N}8$

```
MODULE main
  VAR
    request: boolean;
    state: {ready,busy};
  ASSIGN
    request := FALSE;
    init(state) := ready;
    next(state) :=
      case
        state = ready & request : busy;
        TRUE: {ready,busy};
      esac;
CTLSPEC AG(request -> AF state = busy)
```

Code 6.10: Violation of meta-property $\mathrm{MP_N}10$

$\mathrm{MP_N}10$ No property is vacuously satisfied

A well known problem in formal verification is vacuous satisfaction: a property is vacuously satisfied if that property is satisfied and proved true regardless of whether the model really fulfills what the specifier originally had in mind or not. For example, the LTL property $G(x \rightarrow X(y))$ is vacuously satisfied by any model where x is never true. Vacuity is an indication of a problem in either the model or the property. Several techniques to detect vacuity have been proposed (e.g., [25, 118]) and also tools that perform vacuity detection have been developed (e.g., [82]).

We have already seen in Section 5.2.4 the general strategy to detect vacuity employed in [25, 118]. Let's briefly recall it here. The technique consists in replacing parts of a property and see if this has any effect on the result of the verification. In order to detect vacuity of a property φ, it is sufficient to replace a subformula ψ of φ with *true* or *false* [118], depending on the polarity of ψ in φ. The polarity of a subformula ψ is *positive*, if it is nested in an even number of negations in φ, otherwise is *negative*, and $pol(\psi)$ is a function such that $pol(\psi) = false$ if ψ has positive polarity in φ, and $pol(\psi) = true$ otherwise[1].

The replacement of subformula ψ with ρ in formula φ is denoted as $\varphi[\psi \leftarrow \rho]$.

Definition 6.1. *A property φ is completely/partially vacuous if, for every/some of its atomic proposition ψ, $\mathrm{VC}_\psi = \varphi[\psi \leftarrow pol(\psi)]$ is proved true by the model checking.*

Our tool reports the list of atomic propositions ψ for which VC_ψ is proved true by the model checker.

In Code 6.10, the CTL property is vacuously true for subformula *state = busy*. Indeed the

[1]As in [118] we assume that all the occurrences of the subformula ψ in φ are of *pure polarity*, that is either they are all under an even number of negations (positive polarity), or they are all under an odd number of negations (negative polarity).

CTL formula is *true* regardless of whether *state* is equal to *busy* or not, since *request* is always *false*.

6.3 Meta-property verification by model checking

To verify (or falsify) the meta-properties introduced in the previous section, we translate each instance $\hat{\mathrm{MP}}_{\mathrm{N}}k$ of a meta-property $\mathrm{MP}_{\mathrm{N}}k$ into a CTL property $\hat{\mathrm{MP}}_{\mathrm{N}}^{\mathrm{CTL}}k$. We identify with $\mathrm{MP}_{\mathrm{N}}^{\mathrm{CTL}}k$ the set of all the CTL properties for the meta-property $\mathrm{MP}_{\mathrm{N}}k$. We then build a new NuSMV model M_{MP} obtained by adding to the original model M the set

$$\mathrm{MP}_{\mathrm{N}}^{\mathrm{CTL}} = \bigcup_{k=1}^{10} \mathrm{MP}_{\mathrm{N}}^{\mathrm{CTL}}k$$

that contains the CTL translations of all the instances of all the meta-properties. The verification of the meta-properties is carried out through the model checking of M_{MP}.

The mapping from a meta-property instance $\hat{\mathrm{MP}}_{\mathrm{N}}k$ to a CTL formula $\hat{\mathrm{MP}}_{\mathrm{N}}^{\mathrm{CTL}}k$ is not straightforward, because of *a*) the way CTL properties are verified in NuSMV, *b*) the fact that next_expressions, which can be used in some meta-properties, can not occur in CTL formulae.

a) A CTL property φ is true if and only if φ is true in every initial state of the machine, i.e., given a model R and a property φ,

$$R \models \varphi \quad \text{iff} \quad \forall s_0 \in S_0 \ \ (R, s_0) \models \varphi$$

where S_0 is the set of initial states of R.
The operator *Always*(φ) is translated to $\mathtt{AG}(\varphi)$. Indeed, $M_{MP} \models \mathtt{AG}(\varphi)$ means that, along all paths starting from each initial state, φ is true in every state (globally), which corresponds to the definition of *Always* (see Formula 5.1 in Section 5.3). So:

$$M_{MP} \models \mathit{Always}(\varphi) \quad \Leftrightarrow \quad M_{MP} \models \mathtt{AG}(\varphi) \tag{6.11}$$

Similarly, *InitiallyA*(φ) is translated as φ, since $M_{MP} \models \varphi$ means that in each initial state φ is true, which corresponds to the definition of *InitiallyA* (see Formula 5.3 in Section 5.3). So:

$$M_{MP} \models \mathit{InitiallyA}(\varphi) \quad \Leftrightarrow \quad M_{MP} \models \varphi \tag{6.12}$$

However, the translation of *Sometime*(φ) is not $\mathtt{EF}(\varphi)$, since $M_{MP} \models \mathtt{EF}(\varphi)$ means that there exists at least one path starting from *each* initial state containing a state in which φ is true, while *Sometime* only requires that there exists *at least* an initial state from which φ will eventually hold (see Formula 5.2 in Section 5.3). This means that there are cases in which $\mathtt{EF}(\varphi)$ is false, since not from every initial state φ will eventually be true, while *Sometime*(φ) is true. To prove *Sometime*(φ) we use the following equivalence:

$$M_{MP} \models \mathit{Sometime}(\varphi) \quad \Leftrightarrow \quad M_{MP} \not\models \mathtt{AG}(\neg\varphi) \tag{6.13}$$

that means that *Sometime*(φ) is true if and only if $\mathtt{AG}(\neg\varphi)$ is false. We run the model checker with the property $P = \mathtt{AG}(\neg\varphi)$ and, if a counterexample of P is found, then *Sometime*(φ) holds, while if P is proved true, then *Sometime*(φ) is false.
Similarly, to prove *InitiallyS* (see Formula 5.4 in Section 5.3) we use the equivalence:

$$M_{MP} \models \mathit{InitiallyS}(\varphi) \quad \Leftrightarrow \quad M_{MP} \not\models \neg\varphi \tag{6.14}$$

b) It is possible that a meta-property contains *next_expressions*[2]. In NuSMV such expressions can not be contained in CTL formulae, but they can occur in *invariant specifications.* Invariant

[2]The meta-properties that are defined through the *Always* or the *Sometime* operators can contain the **next** operator (i.e., $\mathrm{MP}_{\mathrm{N}}1$, $\mathrm{MP}_{\mathrm{N}}2$, $\mathrm{MP}_{\mathrm{N}}3$, $\mathrm{MP}_{\mathrm{N}}4$ and $\mathrm{MP}_{\mathrm{N}}5$ when are applied to next assignments).

specifications are propositional formulae which must hold invariantly in the model, and are expressed as "**INVARSPEC** next_expr". They are equivalent to "**CTLSPEC AG** simple_expr" and can be checked by a specialized algorithm during reachability analysis.
In conclusion, all the CTL formulae obtained by the translation described previously, that have the form $\hat{\mathrm{MP}}_{\mathrm{N}}^{\mathrm{CTL}} k = \mathtt{AG}(\varphi)$, with φ containing next_expressions, are checked as invariant specifications. All the other CTL formulae, instead, are checked as CTL specifications.

6.4 NuSMV Model Advisor

We have implemented a prototype tool, available at [138], written in Java to automatize the model review process. The tool is built on top of the NuSVM model checker and required to develop a new parser to represent the structure of a NuSMV specification in terms of Java navigable objects that could be visited to compute the assignment condition functions and to access other internal syntactical specification elements. To the purpose of developing this new parser, the Xtext [171] framework was used. It allows the development of language infrastructures including compilers and interpreters as well as full blown Eclipse-based IDE integration. The user must only provide an EBNF grammar of its language. Starting from this grammar, the XTEXT generator creates a parser, a language meta-model (implemented in EMF) as well as a full-featured Eclipse-based editor.

For our application, we have written the EBNF grammar of NuSMV and through Xtext we have obtained a NuSMV parser. Parsing a model, an EMF model of the NuSMV specification is built, which allows accessing the structure of the specification (otherwise accessible by constructing its abstract syntax tree).

The model advisor works in the following way:

1. the model M one likes to review is parsed by the NuSMV parser provided by the model checker; if M is not parsed correctly the tool does not execute any verification and quits, otherwise it continues as follows;

2. the model M is parsed with our NuSMV parser and an EMF model of M is internally represented;

3. the CTL properties $\mathrm{MP}_{\mathrm{N}}^{\mathrm{CTL}}$ needed for the verification of the meta-properties are built as described in Section 6.3, as well as the NuSMV model $M_{MP} = M + \mathrm{MP}_{\mathrm{N}}^{\mathrm{CTL}}$ obtained from the original specification M with the CTL meta-properties;

4. the tool runs the specification M_{MP} with the model checker NuSMV and reads the output of the execution;

5. it interprets the $\mathrm{MP}_{\mathrm{N}}^{\mathrm{CTL}}$ verification results and builds the meta-properties results that are finally printed (on the screen or on a file).

6.5 Experimental results

We have applied our model review process to three different sets of NuSMV specifications:

- the `NuSMVsrc` set contains the NuSMV examples available in the NuSMV source distribution; some of these examples are also available in the example page on the NuSMV site [137];

- the `Internet` set contains various models that we have found on the Internet: research works, students projects, etc.;

- the `AsmetaSMV` set contains the models obtained with the tool AsmetaSMV, a tool that translates ASM models into NuSMV models (see Section 9.2). This last set of examples was chosen to assess the quality of NUSMV models obtained from models developed using other

Spec Set	# spec.	# rev.	# not rev.	MP_N1	MP_N2	MP_N3	MP_N4	MP_N5
`NuSMVsrc`	63	47	3 - 13	178	230	882	683	44
`Internet`	187	151	30 - 6	209	261	392	351	104
`AsmetaSMV`	34	33	0 - 1	94	121	20	151	34
total	284	231	33 - 20	481	612	1294	1185	182

Spec Set	MP_N6	MP_N7	MP_N8	MP_N9	MP_N10
`NuSMVsrc`	120 (47)	7	3	42	44
`Internet`	2201 (105)	12	8	147	184
`AsmetaSMV`	215 (150)	0	0	1	22
total	2536 (302)	19	11	190	250

Table 6.1: Experimental results and violations found

(high level) formal notations. Indeed, NuSMV is often used as a target language for model checking specifications originally developed using other formal methods. By translating these other models to NuSMV, the NuSMV code might be not efficient and redundancies might be introduced.

The results of our experiments are reported in Table 6.1. It shows the name of the set, the number of models in it, the number of models which we were able to analyse[3] and, for each meta-property, the number of violations we detected.

The most violated meta-property is MP_N6, that is that a variable does not take all the values declared in its type. The high number of violations is also due to the fact that each value not taken is a violation (the number of variables that do not take all their values is shown in round brackets). Simply removing the unused values of the variables type (if they are really not necessary) can dramatically improve the model performances.

The second most violated property is MP_N3, that requires that two conditions are always mutually exclusive. When a couple of conditions $(cond_1, cond_2)$ violates this property, the first condition $cond_1$ masks the second one $cond_2$: sometimes the developer is conscious of this behaviour, but sometimes she is not.

The third most violated property is MP_N4, that requires that the default condition, if specified, is never taken; this meta-property is very strong: developers, indeed, often use the default condition to catch some situations not captured by the previous conditions. We must remember that our meta-properties do not signal errors, but violations of some modeling guidelines that the developer would like to follow.

Finally we would like to underline that the violations of meta-properties MP_N2 and MP_N1 (the third and the fourth most violated meta-properties) signal erroneous models where, respectively, some assignments are never applied and some assignment conditions are never satisfied. Moreover we can notice that all MP_N1 violations are also MP_N2 violations, but not vice versa. Indeed, there are assignments that are never executed, whose conditions are eventually satisfied: these assignments are never executed because their conditions are masked by some previous conditions.

Violations in the `AsmetaSMV` set deserve a particular remark. As expected, the `AsmetaSMV` set contains several violations concerning the minimality of the model, since these models are obtained using NuSMV as target language to model checking ASM models. For example, since all the locations of an ASM function become variables with the same type (the translation of the codomain of the ASM function), it is probable that some of these variables do not take some of their values, since in the original ASM model not all the locations take all the values of the function codomain. Therefore, the high number of MP_N6 violations was expected.

[3]Some models could not be analysed because *a*) they were wrong, that is they did not parse with the NuSMV parser (33 models) *b*) the verification of their meta-properties could have been longer than one hour, the execution time limit we have set (20 models).

However, the violations of property MP_N3 (20 violations in 5 models) is, at a first sight, surprising. The tool AsmetaSMV, indeed, should always produce conditions mutually exclusive. We have discovered that these violations are produced by ASM models containing inconsistent updates, namely parallel updates, in the same state, of the same location (variable in NuSMV) to two different values. This proves that the analysis done at the level of NuSMV can give insights about the high level starting models. In the future, we plan to integrate our NuSMV model advisor with the AsmetaSMV tool, in order to obtain minimal models from the translation of the ASM models and check for model consistency.

6.6 Fault detection capability

An important question about the technique we propose is what kind of faults it can reveal. Although a violation of a meta-property does not necessarily mean that the specification is faulty, it is important to link the automatic analysis we perform to possible faults in the specifications for two reasons: (1) to be sure that the meta-properties actually measure the quality of the specification also in terms of its correctness (which can be ultimately considered as the absence of faults) and (2) to provide useful feedback to the user to suggest, given a violation of a specific meta-property, which kinds of faults can occur in the specification. We have identified the following defects:

- *Over-specification or missing use of variables* is detected by meta-properties like MP_N7 and MP_N8 or by MP_N6 which checks that variable values are all used. These meta-properties aim at detecting faults either of over-specification, i.e., useless details are added to the model, or of omission, i.e., variables that should occur in conditions or expressions but are simply forgotten.
- *Faults in assignments* can be detected by MP_N5. In fact, if a next assignment, when executed, always confirms the value currently assumed by the variable, there is probably an error to fix.
- *Missing or misplaced conditions* can be detected by MP_N1 and MP_N2. Indeed if conditions are placed in a wrong order, then an assignment can be masked and this is signalled by our meta-properties. MP_N3 and MP_N4 try to prevent these kinds of faults by making the conditional assignments independent of the order of the conditions.
- *Wrong or inaccurate properties* are detected by MP_N9 and MP_N10.

In Chapter 8 we evaluate the fault detection capability of the NuSMV model advisor using a technique based on *mutation analysis* (an approach similar to mutation testing [106]).

Chapter 7

Model review of ASMs

We now tackle the problem of automatically reviewing formal specifications given in terms of ASMs (Section 2.1).

The choice of defining a model review process for the ASM formal method is due to several reasons. First, the ASMs are a powerful extension of the Finite State Machines (FSMs), and it has been shown [32] that they capture the principal models of computation and specification in the literature. Therefore, the results obtained for the ASMs can be adapted to other state-transition based formal approaches. Furthermore, thanks to the ASMETA framework (see Section 2.2) and, in particular, to its model checker AsmetaSMV (see Section 9.2), we can easily automatize the proposed approach. Finally, there are several non-trivial specifications on which to test our process.

As done for NuSMV specifications in Chapter 6, we first identify those defects, vulnerabilities, and deviations from standards that a developer can introduce during the modeling activity using the ASMs. Then we define some meta-properties that permit to capture these faults.

We have identified seven meta-properties which use two operators, *Always* and *Sometime*, among those defined in Section 5.3. In order to verify these meta-properties, the ASM model under review is translated into a NuSMV model using the AsmetaSMV tool and the meta-properties are translated into CTL formulae using the technique already described in Section 6.3 for the NuSMV model advisor. Then, the translated model is model checked against the CTL properties to verify if the original ASM model guarantees the meta-properties.

Section 7.1 defines a function, later used in the meta-properties definition, that statically computes the firing condition of a transition rule occurring in the model. Meta-properties that are able to guarantee certain quality attributes of a specification are introduced in Section 7.2. In Section 7.3, we describe how we use AsmetaSMV to check the possible violation of meta-properties. As a proof of concept, in Section 7.4 we report the results of applying our ASM review process to a certain number of specifications, going from benchmark models to test the meta-properties, to ASM models of real case studies of various degree of complexity.

7.1 Rule Firing Condition

In the following we introduce a method to compute, for each rule of the specification under review, the firing condition under which the rule is executed. We introduce a function *Rule Firing Condition* (*RFC*) which returns this condition.

$$RFC : Rules \rightarrow Conditions$$

where *Rules* is the set of the rules of the ASM M under review and *Conditions* are boolean predicates over the state of M. *RFC* can be statically computed as follows.

a) First build a static directed graph, similar to a program control flow graph. Every node of the graph is a rule of the ASM and every edge has label $[u]c$ representing the conditions under which the target rule is executed. c is a boolean predicate and $[u]$ is a sequence of logical assignments of the form $v = t$, being v a variable and t a term. The condition c

Parallel rule R R_1 **par** R_2	$\begin{tikzcd} R \arrow[r, "{[]true}"] \arrow[rd, "{[]true}"] & R_1 \\ & R_2 \end{tikzcd}$
Let rule R **let** $x = t$ **in** R_1	$\begin{tikzcd} R \arrow[r, "{[x=t]true}"] & R_1 \end{tikzcd}$
Macro call rule R $R_m[t_1, .., t_n]$	$\begin{tikzcd} R \arrow[r, "{[x_1=t_1,...,x_n=t_n]true}"] & R_m \end{tikzcd}$

Conditional rule R
if c **then**
R_1
else
R_2
endif
	$\begin{tikzcd} R \arrow[r, "{[]c}"] \arrow[rd, "{[]\neg c}"] & R_1 \\ & R_2 \end{tikzcd}$

Forall rule R **forall** x **in** D **with** a_x **do** R_x	$\begin{tikzcd} R \arrow[r, bend left, "{[x=d_1]a_x}"] \arrow[r, bend right, "{[x=d_n]a_x}"'] & R_x \end{tikzcd}$
Choose rule R **choose** x **in** D **with** a_x **do** R_x	$\begin{tikzcd} R \arrow[r, bend left, "{[x=d_1]a_x}"] \arrow[r, bend right, "{[x=d_n]a_x}"'] & R_x \end{tikzcd}$

Figure 7.1: Schemas for building the graph for the *Rule Firing Condition*

must be evaluated under every logical assignment $v = t$ listed in u. Fig. 7.1 reports how to incrementally build the graph, together with the labels for the edges. By starting from the main rule, the entire graph is built, except for the rules that are never used or are not reachable from the main rule and for which the *RFC* evaluates to *false*. We assume that there are no recursive calls of ASM rules, so the graph is *acyclic*. In general, an ASM rule can call itself (directly or indirectly), but rule recursion is seldom used. However, recursion is still supported in derived functions, which are often used in ASM specifications. For this reason the lack of recursive rules does not prevent to write realistic specifications.

b) Then, to compute the *RFC* for a rule R, start from the rule R and visit the graph backward until the main rule is reached.

The condition $RFC(R)$ is obtained by applying the following three steps. Initially, $R_x = R$ holds.

1. Expand every occurrence of $RFC(R_x)$ by substituting it with the conditions under which R_x is reached, i.e., the labels of the edges entering the node of R_x. If the graph has the schema shown below, one must substitute $RFC(R_x)$ with $[u_1](RFC(R_1) \wedge c_1) \vee \cdots \vee [u_n](RFC(R_n) \wedge c_n)$

$$\begin{tikzcd} R_1 \arrow[rd, "{[u_1]c_1}"'] & R_2 \arrow[d, "{[u_2]c_2}"] & \cdots\cdots & R_n \arrow[lld, "{[u_n]c_n}"] \\ & R_x & & \end{tikzcd}$$

2. Eliminate every logical assignment by applying the following rules:

```
asm exampleForRFC
import StandardLibrary

signature:
    dynamic controlled y: Integer
    dynamic controlled z: Integer

definitions:
    main rule R =
        par
r1:         forall $x in {0,2} with $x < 2 do
r2:             if y < $x then
r3:                 z := y
                endif
r4:         skip
        endpar
```

Code 7.1: Example for the computation of the *Rule Firing Condition*

$$\begin{tikzcd}
R \arrow[r, "true"] \arrow[dr, "true"'] & r1 \arrow[r, bend left, "{[\$x=0]\$x<2}"] \arrow[r, bend right, "{[\$x=2]\$x<2}"'] & r2 \arrow[r, "{[]y<\$x}"] & r3 \\
& r4 & &
\end{tikzcd}$$

Figure 7.2: *RFC* graph of the ASM model shown in Code 7.1

- Distribute the $\vee$ (or) over the $\wedge$ (and):

$$([u_1]A_1 \vee \cdots \vee [u_n]A_n) \wedge B \equiv [u_1](A_1 \wedge B) \vee \cdots \vee [u_n](A_n \wedge B)$$

- Distribute the assignments:

$$[u](A \wedge B) \equiv [u]A \wedge [u]B$$

- Apply the assignments:

$$[u, x = t]A \equiv [u]A[x \leftarrow t]$$

3. Apply again 1 until you reach a rule with no entering edges (main rule).

Example Consider the ASM model shown in Code 7.1 in which y and z are nullary functions of the machine and $\$x$ is a logical variable. The inner rules are labeled for their concise representation in the graph. The *RFC* graph is shown in Fig. 7.2.

To compute the condition under which rule `r3` fires, i.e., $RFC(r3)$, one must perform the following steps:

1. Apply the expansion of $RFC(r3)$:
$$RFC(r3) \equiv RFC(r2) \wedge y < \$x$$
2. Since there is no assignment to eliminate, expand $RFC(r2)$:
$$RFC(r3) \equiv ([\$x = 0](RFC(r1) \wedge \$x < 2) \vee [\$x = 2](RFC(r1) \wedge \$x < 2)) \wedge y < \$x$$
3. Distribute the $\vee$ over the $\wedge$:
$$\begin{aligned} RFC(r3) \equiv\ & [\$x = 0](RFC(r1) \wedge \$x < 2 \wedge y < \$x) \vee \\ & [\$x = 2](RFC(r1) \wedge \$x < 2 \wedge y < \$x) \end{aligned}$$
4. Apply the assignments:

```
main rule r_inc0 =
    par
        l := 1
        l := 2
    endpar
```

Code 7.2: Apparent inconsistent update

```
main rule r_inc1 =
    par
        if cond1 then
            l(a1) := t1
        endif
        if cond2 then
            l(a2) := t2
        endif
    endpar
```

Code 7.3: Inconsistent update less trivial

$RFC(r3) \equiv (RFC(r1) \wedge 0 < 2 \wedge y < 0) \vee (RFC(r1) \wedge 2 < 2 \wedge y < 2)$
$RFC(r3) \equiv (RFC(r1) \wedge y < 0) \vee false$

5. Expand the definition of $RFC(r1)$ which is *true*:
$RFC(r3) \equiv y < 0$

7.2 Meta-properties

We here describe seven meta-properties ($MP_A 1 - 7$) that we have identified for ASMs. All the meta-properties describe attributes that we think that any ASM model should have.

The three categories of model quality attributes introduced in Section 5.3 for state-based formal methods, can be adapted to the ASMs in the following way:

- **Consistency** guarantees that locations (memory units) are never simultaneously updated to different values ($\mathrm{MP_A}1$). This fault is known as *inconsistent update* and must be removed in order to have a correct model.
- **Completeness** requires that every behaviour of the system is explicitly modeled. This enforces explicit listing of all the possible conditions in conditional rules ($\mathrm{MP_A}2$) and the actual updating of controlled locations ($\mathrm{MP_A}7$).
- **Minimality** guarantees that the specification does not contain elements – i.e., transition rules, domain elements, locations, etc. – defined or declared in the model but never used ($\mathrm{MP_A}3$, $\mathrm{MP_A}4$, $\mathrm{MP_A}5$, $\mathrm{MP_A}6$). Minimality of the state requires that only the necessary state functions are introduced ($\mathrm{MP_A}7$).

7.2.1 Meta-property definition

In the following we report the formal definitions of the seven meta-properties that use the logical operators *Always*, *Sometime* defined in Section 5.3 (formulae 5.1 and 5.2).

$\mathrm{MP_A}1$ No inconsistent update is ever performed

An inconsistent update occurs when two updates clash, i.e., they refer to the same location but are distinct (see Def. 2.6). If a location is updated by only one rule, no inconsistent update occurs. Otherwise an inconsistent update is possible.

Let's see two examples. In the example shown in Code 7.2, the same location l is updated to two different values (1 and 2) in two rules having both conditions RFC equal to *true*; in this case, the inconsistent update is apparent. In the example shown in Code 7.3, instead, to prove that the two updates are consistent, one should prove:

$$Always((cond1 \wedge cond2 \wedge a1 = a2) \rightarrow t1 = t2)$$

In general, for every pair of update rules $\hat{R}$ and $\tilde{R}$ of the form $f(\hat{t_1}, \ldots, \hat{t_n}) := \hat{t}$ and $f(\tilde{t_1}, \ldots, \tilde{t_n}) := \tilde{t}$, the property:

```
if x > 0 then
    skip
else
    if x <= 0 then
        skip
    endif
endif
```

Code 7.4: Complete conditional rule

```
if a and b then
    skip
else
    if not a then
        skip
    endif
endif
```

Code 7.5: Incomplete conditional rule

$$Always\left(\left(RFC(\hat{R}) \wedge RFC(\tilde{R}) \wedge \bigwedge_{i=1}^{n} \hat{t_i} = \tilde{t_i}\right) \rightarrow \hat{t} = \tilde{t}\right) \tag{7.1}$$

states that the two updates are never inconsistent. The violation of property 7.1 means that there exists a state in which $\hat{R}$ and $\tilde{R}$ fire, they identify the same location, and $\hat{t} \neq \tilde{t}$.

MP$_\mathrm{A}$2 Every conditional rule must be complete

In a conditional rule R = **if** c **then** R_{then} **endif**, without the *else* branch, the condition c must be true if R is evaluated. Therefore, in a nested conditional rule, if one does not use the else branch, the last condition must be true.

In Code 7.4 the inner conditional rule is complete, since, if guard $x > 0$ of the outer conditional rule is false, the guard $x \leqslant 0$ is true. In Code 7.5, instead, the inner conditional rule is incomplete, since, if guard *a and b* of the outer conditional rule is false with a being *true* and b being *false*, then no branch in the conditional statements is chosen. Property

$$Always(RFC(R) \rightarrow c) \tag{7.2}$$

states that, when the conditional rule R is executed, its condition c is evaluated to true. A violation of property 7.2 means that there exists a behaviour of the system that satisfies $RFC(R) \wedge \neg c$ but it is not explicitly captured by the model.

Corollary 1: Every Case Rule without otherwise must be complete Since the case rule can be reduced, by definition [32], to a series of conditional rules, the computation of RFC is straightforward. The meta-property MP$_\mathrm{A}$2 is applied to case rules as follows. Let

```
switch $t$
    case $t_1$ : $R_1$
    ...
    case $t_n$ : $R_n$
endswitch
```

be a case rule R. Its completeness is given by the following property:

$$Always\left(RFC(R) \rightarrow \bigvee_{i=1}^{n} t = t_i\right) \tag{7.3}$$

The violation of the property 7.3 means that there exists a state in which the case rule R is executed and no branch is taken.

MP$_\mathrm{A}$3 Every rule can eventually fire

Let R be a rule of our ASM model; to verify that R is eventually executed, we must prove the following property:

$$Sometime(RFC(R)) \tag{7.4}$$

If the property is proved false, it means that rule R is contained in an unreachable model fragment.

```
if x > 0 then
    if x < 0 then
        skip
    endif
endif
```

Code 7.6: Conditional rule with guard never true

Corollary 2: Every condition in a conditional rule is eventually evaluated to true (and false if the else branch is given) For every conditional rule, $\mathrm{MP_A3}$ requires that there exists a path in which its guard is eventually true and, if the else is given, also a path in which its guard is eventually false. In the example in Code 7.6 the guard of the inner conditional rule is never true.

Let $Q = \textbf{if}\, c\, \textbf{then}\ R_{then}\ [\textbf{else}\, R_{else}]\ \textbf{endif}$ be a conditional rule. The property 7.4 becomes, for the **then** and the **else** branches, respectively:

$$Sometime(RFC(Q) \wedge c) \tag{7.5}$$

$$Sometime(RFC(Q) \wedge \neg c) \tag{7.6}$$

$\mathrm{MP_A4}$ No assignment is always trivial

An update $l := t$ is trivial [87] if l is already equal to t, even before the update is applied. This property requires that each assignment which is eventually performed, will not be always trivial. Let $R = \ l := t$ be an update rule. Property

$$Sometime(RFC(R)) \rightarrow Sometime(RFC(R) \wedge l \neq t) \tag{7.7}$$

states that, if eventually updated, the location l will be updated to a new value at least in one state. Note that the more simple property $Sometime(RFC(R) \wedge l \neq t)$ would be false if the update is never performed.

$\mathrm{MP_A5}$ For every domain element e there exists a location which has value e

Every domain element should be used at least once as location value. In the example of Code 7.7, the element *OUTOFMONEY* of the domain *State* is never used. To check that a domain element $e_j \in D$ is used as location value, if $l_1, \dots, l_n$ are all the locations (possibly defined by different function names) taking value in the domain D, the property

$$Sometime\left(\bigvee_{i=1}^{n} l_i = e_j\right) \tag{7.8}$$

states that at least a location once takes the value e_j. Note that this property must be restricted to domains that are only function co-domains: if the domain D is used as domain of an n-ary function with $n > 0$, all its elements have to be considered useful, even if property 7.8 would be false for some $e_j \in D$. Otherwise, if property 7.8 is false, the element e_j may be wrongly removed from the domain.

$\mathrm{MP_A6}$ Every controlled function can take any value in its co-domain

Every controlled function is assigned at least once to each element in its co-domain; otherwise it could be declared over a smaller co-domain. Let $l_1 \dots l_n$ be the locations of a controlled function f with co-domain $D = \{e_1, \dots, e_m\}$. Property

```
asm ATM
import StandardLibrary

signature:
    enum domain State = {AWAITCARD | AWAITPIN | CHOOSE | OUTOFSERVICE | OUTOFMONEY}
    dynamic controlled atmState: State
    dynamic controlled atmInitState: State
    dynamic controlled atmErrState: State
    dynamic monitored pinCode: Integer

    main rule r_Main =
        par
            if(atmState = atmInitState) then
                atmState := AWAITPIN
            endif
            if(atmState=AWAITPIN) then
                atmState := CHOOSE
            endif
            if(atmState=CHOOSE) then
                atmState := AWAITCARD
            endif
        endpar

default init s0:
    function atmInitState = AWAITCARD
    function atmErrState = OUTOFSERVICE
    function atmState = atmInitState
```

Code 7.7: Over-specified ATM model

$$Sometime\left(\bigvee_{i=1}^{n} l_i = e_1\right) \wedge Sometime\left(\bigvee_{i=1}^{n} l_i = e_2\right) \wedge \ldots \wedge Sometime\left(\bigvee_{i=1}^{n} l_i = e_m\right) \tag{7.9}$$

states that f takes all the values of its co-domain D. Actually, in order to discover what values of the co-domain are never taken, each *Sometime* must be checked independently.

MP$_A$7 Every controlled location is updated and every location is read

This meta-property is obtained combining the results of the previous meta-properties and a static inspection of the model. It checks if a location is useful and if it has been declared correctly (e.g., a controlled function could be static). The meta-property is defined by Table 7.1.

controlled	**initialised**	**updated**	**always trivial update**	**read**	**Possible actions**
false	N/A	N/A	N/A	false	remove - read somewhere
true	-	false	N/A	false	remove - read somewhere and/or add an update
true	true	false	N/A	true	declare static - add an update
true	true	true	true	-	declare static

Table 7.1: Indicators used in the verification of MP$_A$7

Given a location, the static inspection of the code must establish whether it is a controlled location, or it is a monitored/static/derived location (column *controlled*). Also by static inspection, it must be discovered if the location is initialised (column *initialised*); this check is applicable only to controlled locations (N/A when *controlled* is false).

Moreover, through $\mathrm{MP_A}3$ we check if the location is updated (column *updated*): it is applicable only to controlled locations. We check that there exists at least an update rule in which the location is on the left-hand side.

By $\mathrm{MP_A}4$ we check if the update is always trivial (column *always trivial update*): it is applicable only to controlled locations which have actually been updated.

Finally, by $\mathrm{MP_A}3$ we check if the location is read in at least one state.

The combination of the results of the single checks can suggest different possible solutions to fix the model. Let's see, as example, the second row of the table. If a controlled function is never updated and never read, no matter it is initialised or not, we can think of two possible sources of the error and provide two possible solutions:

a) the location is useless and the corresponding function can be removed (only if all the locations of the same function present the same problem),

b) the modeler forgot to specify some behaviours and so the location should be read and/or updated somewhere in the model.

Let's see the application of the meta-property to a simple example. In the model shown in Code 7.7 the monitored location *pinCode* is never read; this means that it could be removed or that a part of the model is missing in which the location should be read. The controlled *atmErrState* location is initialised, but never updated nor read; it could be removed or it should be used in some unspecified part of the model. The controlled *atmInitState* location is initialised, read, but never updated; it could be declared static.

7.3 Meta-property verification by model checking

To verify (or falsify) the meta-properties introduced in the previous section, we use the AsmetaSMV tool (see Section 9.2) which is able to prove temporal properties of ASM specifications by using the model checker NuSMV. The ASM specification M is translated to a NuSMV machine M_{NuSMV} representing the Kripke structure which is model checked to verify a given temporal property.

We here use CTL to express the properties to be verified by NuSMV. The technique used to translate meta-properties into CTL formulae is the same described in Section 6.3 for the NuSMV model advisor.

7.4 Experimental results

We have implemented a prototype tool, available at [13], that has allowed us to apply our model review process to three different sets of ASM specifications:

- the `Bench` set contains only the benchmarks we have explicitly designed to expose the violations of the introduced meta-properties;
- the `AsmRep` set contains models taken from the ASMETA repository which are also available at [13]. Many ASM case studies of various degree of complexity and several specifications of classical software engineering systems (like ATM, Dining Philosophers, Lift, etc.) are included in `AsmRep`.
- the `Stu` set contains the models written by the students of a master course in which the ASM method is taught.

The results of our experiments are reported in Table 7.2 which shows the name of the set, the number of models in it, the total number of rules in those models, the number of violations we detected, and the violations found in terms of meta-properties.

As expected our tool was able to detect all the violations in the benchmarks. The student projects contained several faults, most regarding the model minimality but also some inconsistencies which were not detected by model simulation. We found also several violations in the models

Spec Set	# spec.	# rules	# violations	violated $\mathrm{MP_A}$s (# violations)
`Bench`	21	384	61	All
`AsmRep`	18	506	29	$\mathrm{MP_A}$4(11), $\mathrm{MP_A}$6(8), $\mathrm{MP_A}$5(5) $\mathrm{MP_A}$7(4), $\mathrm{MP_A}$3(1)
`Stu`	6	172	38	$\mathrm{MP_A}$7(11), $\mathrm{MP_A}$5(9), $\mathrm{MP_A}$6(9), $\mathrm{MP_A}$1(3), $\mathrm{MP_A}$3(3), $\mathrm{MP_A}$4(3)

Table 7.2: Experimental results and violations found

of `AsmRep`, all of them regarding model minimality. Note that not all the models in `AsmRep` could be analysed, since AsmetaSMV does not support all the AsmetaL constructs and it can analyse only finite models.

Chapter 8

Assessing the fault detection capability of a model review technique

In Chapters 6 and 7 we have presented two model review techniques for, respectively, NuSMV specifications and ASMs. These techniques identify model attributes and characteristics that any model should assure, independently from the particular model to analyse. These quality attributes are expressed as formal predicates, called *meta-properties*: they can be assumed as measures of model quality.

The main aim of model review is to check if models are easy to develop, maintain, and enhance, not to detect behavioural faults. However, also the (stylistic) violations identified by a model reviewer can bring to behavioural faults. So, it would be of great importance to be able to assess the capability of tools for static checking (like model reviewers) in detecting errors in models, especially actual behavioural faults. However, while testing fault detection capability has been extensively studied, since testing explicitly targets behavioural faults, the fault detection capability of static analysis has not been studied with the same strength. For this reason, we propose a way to assess the fault detection capability of static model review by using *mutation*.

Mutation is a well known technique in the context of software code, and program mutation consists in introducing small modifications into program code such that these simple syntactic changes, called *mutations*, represent typical mistakes that programmers often make. These faults are deliberately seeded into the original program in order to obtain a set of faulty programs called *mutants*. Program mutation is almost always used in combination with testing. High quality test suites should be able to distinguish the original program from its mutants, i.e., to detect the seeded faults. The history of mutation testing can be traced back to the 70s as reported by [106]. Mutation testing has been applied to many programming languages and for any sort of domain application. More recently, it has been applied to specifications like FSMs [64], Petri nets [65], Statecharts [66], Object-Z specifications [127], Estelle specifications [56], instead of programs.

We propose using mutation analysis in combination with static model review instead of testing, and we operate at specification level instead of at code level. We evaluate the fault detection capability of the NuSMV model advisor, but the idea behind the presented approach could be used to evaluate also the ASM model advisor or other model review techniques.

The idea is quite simple: can we use the mutation of NuSMV models to assess the quality of the analysis performed by the model advisor? A static analysis like that performed by our model advisor, to be really useful in practice, should be able to distinguish between correct specifications and faulty ones, in a similar way as tests are able to kill mutants in mutation testing. Note that a model advisor is designed to enforce a set of style and consistency rules with the main goal of increasing model qualities like maintainability and readability, while it does not target behavioural correctness.

Section 8.1 presents a NuSMV model we use as running example. Section 8.2 introduces a set of mutation operators for NuSMV, representing possible mistakes designers can make. Each operator produces a set of mutated specifications, which however could be equivalent to, i.e., behave as, the original one (Section 8.3). Equivalent mutants pose a challenge to our method,

```
MODULE main
VAR
    hour: 0..23;
    hour12: 1..12;
    amPm: {AM, PM};
ASSIGN
    init(hour) := 0;
    next(hour) := (hour + 1) mod 24;
    hour12 :=
        case
            hour in {0, 12}: 12;
            !(hour in {0, 12}): hour mod 12;
        esac;
    amPm :=
        case
            hour < 12: AM;
            hour >= 11: PM;
        esac;

CTLSPEC NAME pmOK := AG(hour > 11 -> amPm = PM);
```

Code 8.1: NuSMV model of a clock (only hours)

since they do not represent actual faults. A novel technique for checking equivalence between Kripke structures is presented in Section 8.4, and extended to NuSMV models in Section 8.5. Section 8.6 presents the process we have devised to combine the use of mutation and static model review. This process is able to classify mutants in four cases: it considers if a mutant has been killed or not, and if it is equivalent or not. The desired outcome would be to have only killed not equivalent mutants and not killed equivalent ones. We present a series of experiments and statistical analyses in Section 8.7. We are able to assess the quality of our method by measuring its *sensitivity*, *precision*, and *accuracy*.

8.1 Running example

Code 8.1 shows the NuSMV specification that we use as running example throughout the chapter to describe our approach. It is a model of a clock that memorizes only the hours: the variable *hour* provides the hour in the 24-hour format, whereas variables *hour12* and *amPm* provide the hour in the 12-hour format. Variable *hour* is initialised to zero and it is incremented of a unit (modulo 24) in each transition from a state to the next one. The values of *hour12* and *amPm* are defined based on the value of *hour*. A CTL property checks that, in each state, if *hour* is greater than 11, then *amPm* is *PM*; the property identifier is *pmOK*.

Model review We checked the model in Code 8.1 using our model advisor. The assignment of variable *amPm* violates meta-property MP_N3 (see Section 6.2.1), since the conditions $hour < 12$ and $hour >= 11$ are not mutually exclusive: both conditions are true when variable *hour* takes value 11. Note that, however, the definition of variable *amPm* is correct since, when *hour* takes value 11, *amPm* correctly assumes value *AM* (because the first branch of the case expression whose condition evaluates to true is taken). In this case, the violation of the meta-property has indicated a stylistic defect, not a real fault.

8.2 Mutation operators for NuSMV models

In order to generate *mutated specifications* (or *mutants*), we must define some *mutation operator*s, i.e., rules that specify syntactic variations of the specification [2]. The standard way is to derive the operators directly from *fault classes*, namely errors that can be introduced by the developer in the specification: typical fault classes are those defined by Kuhn in [117].

We have identified a group of mutation operators: some of them are the usual ones described in literature (e.g., LOR, SA0), others are more specific operators tailored on NuSMV specifications

```
MODULE main
VAR
    hour: 0..23;
    hour12: 1..12;
    amPm: {AM, PM};
ASSIGN
    init(hour) := 0;
    next(hour) := (hour + 1) mod 24;
    hour12 :=
        case
            hour in {0, 12}: 12;
            !(hour in {0, 12}): hour mod 12;
        esac;
    amPm :=
        case
            hour >= 11: PM;
            hour < 12: AM;
        esac;

CTLSPEC NAME pmOK := AG(hour > 11 -> amPm = PM);
```

Code 8.2: Mutation of the NuSMV model in Code 8.1 – Swapped branches

(e.g., MB, SB). Mutation operators can be classified as follows:

- *Structure mutation operators* modify the structure of the specification:
  - *Missing Branch* (MB): in a case expression one of the branches is removed.
  - *Swapped Branches* (SB): in a case expression two branches are swapped. Given a case expression with n branches, it produces $^{n\cdot(n-1)}/_2$ mutants. The model in Code 8.2 is a mutant of that in Code 8.1, obtained by swapping the two branches of the case expression of the assignment of variable *amPm*.
  - *Missing Definition* (MD): a variable assignment (simple, init or next) is removed from an ASSIGN section.
  - *Missing TRANS/INIT/INVAR* (MTC/MIC/MINC): a TRANS/INIT/INVAR constraint is removed from the specification.
- *Expression mutation operators* modify expressions:
  - *Expression Negation* (EN): an expression is replaced by its negation.
  - *Logical Operator Replacement* (LOR): a logic operator (&, |, →, ↔ *xor*, *xnor*) is replaced by another logic operator.
  - *Mathematical Operator Replacement* (MOR): a mathematical operator (+, −, ∗, /, *mod*) is replaced by another one.
  - *Relational Operator Replacement* (ROR): a relational operator (=, ! =, <, <=, >, >=) is replaced by another one. The model in Code 8.3 is a mutant of that in Code 8.1, obtained by replacing, in the second condition of the case expression of the assignment of variable *amPm*, the operator >= with the operator >.
  - *Stuck-At 0/1* (SA0, SA1): a boolean expression is replaced by the value FALSE/TRUE.
  - *Associative Shift* (AS): in an expression, operators precedence is changed introducing and/or removing parentheses (e.g., $(a \mid b)$ & c is mutated in $a \mid (b$ & $c))$.
- *Value mutation operators* modify the occurrence of numerical or enumerative values:

```
MODULE main
VAR
  hour: 0..23;
  hour12: 1..12;
  amPm: {AM, PM};
ASSIGN
  init(hour) := 0;
  next(hour) := (hour + 1) mod 24;
  hour12 :=
    case
      hour in {0, 12}: 12;
      !(hour in {0, 12}): hour mod 12;
    esac;
  amPm :=
    case
      hour < 12: AM;
      hour > 11: PM;
    esac;

CTLSPEC NAME pmOK := AG(hour > 11 -> amPm = PM);
```

Code 8.3: Mutation of the NuSMV model in Code 8.1 – Relational Operator Replacement

– *Enumeration Replacement* (ER): an enumeration constant is replaced by another enumeration constant belonging to the same domain.
– *Number Replacement* (NR): an integer constant n is replaced by the integer constant $n + 1$ or $n - 1$.
– *Digit Replacement* (DR): given an integer constant n whose decimal representation is $n_m \dots n_0$, a digit n_i (with $i = 0, \dots, m$) is replaced with a different digit (e.g., 98 is mutated in 92). For each integer constant, it produces $9 \cdot (m + 1)$ mutants.

All the mutation operators previously introduced are applied to the ASSIGN, DEFINE, TRANS, INIT and INVAR sections; in this work we do not mutate neither the variable declaration nor the properties specifications.

Since we do not modify neither variable domains nor modules instances, all the mutant specifications we produce have the same state space of the original specification; the mutations we apply can just change the transition relation, the set of reachable states and the set of initial states.

Although the operators introduce only single mutations (first order), they can be applied in sequence in order to obtain higher order mutants [105].

In the rest of the thesis we will identify mutants with the acronym of the mutation operator that generates them.

Note that the mutation operators are voluntarily not directly related to the properties that are addressed by the NuSMV model advisor since they model faults, while meta-properties refer to quality attributes of the models. As already emphasized before, the model advisor is not designed to target behavioural faults.

In the following we will use the function

$$orig : Muts \rightarrow OrigSpecs \tag{8.1}$$

that, given a mutant, retrieves its original specification.

8.3 Equivalent mutants

When a mutant behaves like the original model, it is said *equivalent*. Most mutation operators can produce equivalent mutants, which pose a challenge, since they do not represent actual faults and can not be detected by observing the behaviour of the specification. The problem of functionally equivalent mutants is well-known in mutation testing [2, 85]: an equivalent mutant does not change the semantics of the program. Since the semantics of the program is unchanged, it is impossible (and useless) to write a test that captures it.

The model in Code 8.2 is a non-equivalent mutant of that in Code 8.1, since the seeded mutation (the swapping of the case expression branches in the assignment of variable *amPm*) has changed the behaviour of the model. In fact, when variable *hour* takes value 11, the variable *amPm* takes value *AM* in Code 8.1, whereas it takes value *PM* in Code 8.2.

The model in Code 8.3, instead, is an equivalent mutant of that in Code 8.1, since the seeded mutation (the replacement of the relational operator >= with operator > in the second condition of the case expression of the assignment of variable *amPm*) has not changed the behaviour of the model. In fact, in Code 8.1 the condition *hour* >= 11 is checked only if *hour* is greater than 11, since the previous branch is not taken if condition *hour* < 12 is false. So, replacing the second condition with *hour* > 11 does not affect the behaviour of the model.

Although detecting equivalent mutants is in general an undecidable problem [2] and, when possible, is a time-consuming activity [85] and difficult to automatize, we have devised a technique able to discover NuSMV equivalent mutants. We first present a technique to discover equivalence between Kripke structures in Section 8.4. Then, in Section 8.5, since NuSMV models are a form of Kripke structures, we apply the technique to discover equivalence between NuSMV models.

8.4 Equivalence checking for Kripke structures

In Section 3.1.2 (Theorem 3.1) we have reported the conditions under which two Kripke structures K_1 and K_2, with the same set of atomic propositions, are equivalent.

We here show that the problem of checking the equivalence of K_1 and K_2 (i.e., proving properties 3.1 – 3.4 of Theorem 3.1) can be reduced to the problem of proving some properties over a new *merging* Kripke structure K_{12} derived from K_1 and K_2. In Section 8.4.1 we show how to build K_{12}, and in Section 8.4.2 we introduce a new theorem that establishes the equivalence between K_1 and K_2 based on the verification of some properties in K_{12}.

8.4.1 Construction of the merging Kripke structure

Let $K_1 = \langle S_1, S_1^0, T_1, \mathcal{L}_1 \rangle$ and $K_2 = \langle S_2, S_2^0, T_2, \mathcal{L}_2 \rangle$ be two Kripke structures with the same set of atomic propositions AP.

Let $K_{12} = \langle S_{12}, S_{12}^0, T_{12}, \mathcal{L}_{12} \rangle$ be a Kripke structure built upon K_1 and K_2, satisfying the following conditions.

C1: condition over the states S_{12}. There exist two *projection* functions:

$$\sigma_1 \colon S_{12} \to S_1 \qquad\qquad \sigma_2 \colon S_{12} \to S_2$$

such that

$$\forall s_1 \in S_1, \forall s_2 \in S_2, \exists s_{12} \in S_{12}\ [\sigma_1(s_{12}) = s_1 \wedge \sigma_2(s_{12}) = s_2]$$

C2: condition over the initial states S_{12}^0.

$$\forall s \in S_{12}\ \left[s \in S_{12}^0 \iff \left(\sigma_1(s) \in S_1^0 \wedge \sigma_2(s) \in S_2^0\right)\right]$$

C3: condition over the transition relation T_{12}.

$$\forall s \in S_{12}, \forall s' \in S_{12}\ [s' \in next_{K_{12}}(s) \iff (\sigma_1(s') \in next_{K_1}(\sigma_1(s)) \wedge \sigma_2(s') \in next_{K_2}(\sigma_2(s)))]$$

8.4.1.1 Corollary

A state $s \in S_{12}$ is reachable in K_{12} iff its projections $\sigma_1(s)$ and $\sigma_2(s)$ are, respectively, reachable in K_1 and K_2.

$$isReach_{K_{12}}(s) \iff isReach_{K_1}(\sigma_1(s)) \wedge isReach_{K_2}(\sigma_2(s))$$

Proof. Let's assume that

$$\exists s \in S_{12}\ [isReach_{K_{12}}(s) \wedge \neg isReach_{K_2}(\sigma_2(s))]$$

If s is reachable in K_{12}, it means that there exists a path $\pi = s_1, \ldots, s_n \in \Pi^0_{12}$ such that $s_n = s$.

Let's now consider the sequence of states $\sigma_2(s_1), \ldots, \sigma_2(s_n)$ in K_2. By the assumption made at the beginning of the proof, we know that the projection of state s_n is not reachable in K_2, i.e., $\neg isReach_{K_2}(\sigma_2(s_n))$[1]; this means that $\exists i \in [1, n-1]\colon \sigma_2(s_{i+1}) \notin next(\sigma_2(s_i))$. But this contradicts condition **C3** on the construction of T_{12}. A similar contradiction is achieved if, at the beginning of the proof, we suppose that $\exists s \in S_{12}\ [isReach_{K_{12}}(s) \wedge \neg isReach_{K_1}(\sigma_1(s))]$. □

8.4.2 Conditions on the merging Kripke structure for assessing the equivalence

We here introduce some predicates to investigate the relation between the states of K_{12} and the states of K_1 and K_2. Then, we use them to formulate a theorem (derived from Theorem 3.1) in which the equivalence between K_1 and K_2 is established checking some properties in K_{12}.

Definition 8.1 (Equivalence of the projections). *We say that a state $s \in S_{12}$ is* labelly equivalent *iff the labels of the two projections are equivalent, i.e,*

$$le(s) \triangleq \mathcal{L}_1(\sigma_1(s)) = \mathcal{L}_2(\sigma_2(s))$$

We say that two states $s, s' \in S_{12}$ are labelly equivalent *with respect to the projection σ_i ($i = 1, 2$) iff the labels of their projections σ_i are equivalent, i.e,*

$$le_1(s, s') \triangleq \mathcal{L}_1(\sigma_1(s)) = \mathcal{L}_1(\sigma_1(s')) \qquad le_2(s, s') \triangleq \mathcal{L}_2(\sigma_2(s)) = \mathcal{L}_2(\sigma_2(s'))$$

Definition 8.2 (Mirror state). *For all states $s \in S_{12}$ we define the predicate* mirror *as:*

$$mirror(s) \triangleq le(s) \rightarrow \forall s' \in next(s) \begin{bmatrix} \exists s'' \in next(s)\ [le(s'') \wedge le_1(s', s'')] \wedge \\ \exists s''' \in next(s)\ [le(s''') \wedge le_2(s', s''')] \end{bmatrix}$$

Theorem 8.1 (Equivalence between K_1 and K_2). *K_1 and K_2 are equivalent iff the following properties*

$$\forall s \in S^0_{12},\ \exists s' \in S^0_{12}\ [le(s') \wedge le_1(s, s')] \tag{8.2}$$

$$\forall s \in S^0_{12},\ \exists s'' \in S^0_{12}\ [le(s'') \wedge le_2(s, s'')] \tag{8.3}$$

$$\forall s \in reach(K_{12})\ [mirror(s)] \tag{8.4}$$

hold in K_{12}.

8.5 Equivalence checking for NuSMV models

We here introduce a technique that permits to check if two NuSMV models that have the same state space are equivalent. Since NuSMV models are a form of Kripke structures (see Def. 3.10 in Section 3.2), we adapt the technique shown in Section 8.4 for checking the equivalence between Kripke structures.

In Section 8.5.1 we show the result of mutating a NuSMV model, using the mutation operators previously introduced. In Section 8.5.2 we show how to build a merging NuSMV model starting from a NuSMV model and one of its mutant. Finally, in Section 8.5.3, we describe how to check the equivalence between the original NuSMV model and its mutant by proving some CTL properties over the merging model.

[1] $s = s_n$

8.5.1 Mutating a NuSMV model

As we have seen in Section 8.2, the mutations we apply can change the initial assignments and/or the next state assignments of a set of variables, that is the way in which their initial/next value is calculated. Given a NuSMV model $M_o = \langle S_o, S_o^0, T_o \rangle$, when we apply a mutation, we obtain a model $M_m = \langle S_m, S_m^0, T_m \rangle$ with the same state space and the same variables, i.e., $S_o = S_m$ and $var(M_o) = var(M_m)$, but, maybe, with a different transition relation T_m and/or a different set of initial states S_m^0. If $S_o^0 = S_m^0 \wedge T_o^0 = T_m^0$, the two models are equivalent, otherwise they are not equivalent.

Partitioning of the variables We decompose the variables $var(M_o)$ in subsets, depending on the fact that they are *affected* by the mutation or not:

- $MV = \{\tilde{v}_1, \ldots, \tilde{v}_k\}$ is the set of variables whose initial/next assignment has been mutated. Let $\tilde{D}_1, \ldots, \tilde{D}_k$ be their domains.
- $DV = \{v_{k+1}, \ldots, v_n\}$ is the set of all non mutated variables of M_o upon which the value of some mutated variables depends on, i.e., $v \in DV$ iff there exists a variable $\tilde{v} \in MV$ whose value in some state is determined according to the value of v in the current/previous state. Let $D_{k+1}, \ldots, D_n$ be their domains. We require that $MV \cap DV = \varnothing$.
- IN is the set of variables that are not considered in the evaluation of the value of any variable in MV and that are not mutated.

8.5.2 Merging model

Given the NuSMV models M_o and M_m, we define the *merging* model $M_e = \langle S_e, S_e^0, T_e \rangle$ as the NuSMV implementation of the merging Kripke structure described in Section 8.4.1. M_e is built as follows:

- $var(M_e) = MV \cup MV' \cup DV$, where $MV' = \{\tilde{v}'_1, \ldots, \tilde{v}'_k\}$ is the set of renamed copies of variables in MV. Their domains are the same of the variables in MV, i.e., $\tilde{D}_1, \ldots, \tilde{D}_k$. There exists a bijective function

$$mut : MV \to MV'$$

  such that $\forall \tilde{v}_i \in MV(mut(\tilde{v}_i) = \tilde{v}'_i)$.
- The initial state assignments of variables in $MV \cup DV$ are those defined in M_o, while variables in MV' have initial assignments as in M_m.
- The next state assignments of variables in $MV \cup DV$ are those defined in M_o, while variables in MV' have next state assignments as in M_m.

Note that variables in IN are not considered in the merging model since they are not useful for assessing the equivalence. Identifying variables of IN is a variant of the *cone of influence* technique [51].

Given a state $s \in S_e$, the interpretation of variables $MV \cup DV$ identifies a set of states in S_o, i.e.,

$$origModelStates(s) = \left\{ s_o \in S_o \colon \bigwedge_{i=1}^{k} [\![\tilde{v}_i]\!]_s = [\![\tilde{v}_i]\!]_{s_o} \wedge \bigwedge_{j=k+1}^{n} [\![v_j]\!]_s = [\![v_j]\!]_{s_o} \right\}$$

If $IN = \varnothing$, then $|origModelStates(s)| = 1$ for each $s \in S_e$.

In the same way, given a state $s \in S_e$, the interpretation of variables $MV' \cup DV$ identifies a set of states in S_m, i.e.,

```
MODULE main
VAR
    hour: 0..23;
    amPm: {AM, PM};
    amPmMut: {AM, PM};
ASSIGN
    init(hour) := 0;
    next(hour) := (hour + 1) mod 24;
    amPm :=
        case
            hour < 12: AM;
            hour >= 11: PM;
        esac;
    amPmMut :=
        case
            hour >= 11: PM;
            hour < 12: AM;
        esac;
```

Code 8.4: Merging model of models in Codes 8.1 and 8.2

$$mutModelStates(s) = \left\{ s_m \in S_m \colon \bigwedge_{i=1}^{k} [\![\tilde{v}'_i]\!]_s = [\![\tilde{v}'_i]\!]_{s_m} \wedge \bigwedge_{j=k+1}^{n} [\![v_j]\!]_s = [\![v_j]\!]_{s_m} \right\}$$

Let's see now an example of merging model. Code 8.4 is the merging model to use for checking the equivalence between Codes 8.1 (the original model) and 8.2 (the mutant). The set of variables of the merging model is composed as follows: $MV = \{amPm\}$ and $MV' = \{amPmMut\}$ since the mutation affects the definition of variable *amPm*, and $DV = \{hour\}$ because the definition of variable *amPm* depends on variable *hour*. The transitions relation of variables *amPm* and *hour* are those specified in the original model (Code 8.1), whereas the transition relation of variable *amPmMut* has been derived from the transition relation of variable *amPm* in the mutated model (Code 8.2). Note that variable *hour12* is not included in the merging model because its transition relation has not been mutated and it is not involved in the definition of a mutated variable (i.e., $IN = \{hour12\}$).

Let's now introduce some definitions useful for identifying the states of the merging model as tuple of values.

Definition 8.3 (Initial state as tuple of values)**.** *Let IS be the set of tuples of values of the variables in the initial states S_e^0, i.e.,*

$$IS = \left\{ \begin{array}{l} \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, \tilde{d}'^{k}_{i=1} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right) : \\ \exists s \in S_e^0 \left[\bigwedge_{i=1}^{k} \left(\tilde{d}_i = [\![\tilde{v}_i]\!]_s \wedge \tilde{d}'_i = [\![\tilde{v}'_i]\!]_s\right) \wedge \bigwedge_{j=k+1}^{n} d_j = [\![v_j]\!]_s \right] \end{array} \right\}$$

Let's also define the projections of IS over the original and the mutated models as

$$IS_o = \left\{ \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right) : \exists s \in S_e^0 \left[\bigwedge_{i=1}^{k} \tilde{d}_i = [\![\tilde{v}_i]\!]_s \wedge \bigwedge_{j=k+1}^{n} d_j = [\![v_j]\!]_s \right] \right\}$$

$$IS_m = \left\{ \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right) : \exists s \in S_e^0 \left[\bigwedge_{i=1}^{k} \tilde{d}_i = [\![\tilde{v}'_i]\!]_s \wedge \bigwedge_{j=k+1}^{n} d_j = [\![v_j]\!]_s \right] \right\}$$

By definition of IS, IS_o and IS_m, it holds that

$$\forall \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, \tilde{d}'^{k}_{i=1} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right)$$
$$\left[\left(\tilde{d}_{i=1}^{k}, \tilde{d}'^{k}_{i=1}, d_{j=k+1}^{n}\right) \in IS \iff \left(\left(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}\right) \in IS_o \wedge \left(\tilde{d}'^{k}_{i=1}, d_{j=k+1}^{n}\right) \in IS_m\right)\right]$$

Definition 8.4 (Next state as tuple of values). *Let $NS(s)$ be the set of tuples of values of the variables in the next states of $s \in S_e$, i.e.,*

$$NS(s) = \left\{ \begin{array}{l} \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, \tilde{d}'^{k}_{i=1} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right) : \\ \exists s' \in next(s) \left[\bigwedge_{i=1}^{k} \left(\tilde{d}_i = [\![\tilde{v}_i]\!]_{s'} \wedge \tilde{d}'_i = [\![\tilde{v}'_i]\!]_{s'}\right) \wedge \bigwedge_{j=k+1}^{n} d_j = [\![v_j]\!]_{s'}\right] \end{array} \right\}$$

Let's also define the projections of $NS(s)$ over the original and the mutated models as

$$NS_o(s) = \left\{\left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right) : \exists s' \in next(s) \left[\bigwedge_{i=1}^{k} \tilde{d}_i = [\![\tilde{v}_i]\!]_{s'} \wedge \bigwedge_{j=k+1}^{n} d_j = [\![v_j]\!]_{s'}\right]\right\}$$
$$NS_m(s) = \left\{\left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right) : \exists s' \in next(s) \left[\bigwedge_{i=1}^{k} \tilde{d}_i = [\![\tilde{v}'_i]\!]_{s'} \wedge \bigwedge_{j=k+1}^{n} d_j = [\![v_j]\!]_{s'}\right]\right\}$$

By definition of NS, NS_o and NS_m, it holds that

$$\forall s \in S_e, \forall \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, \tilde{d}'^{k}_{i=1} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right)$$
$$\left[\left(\tilde{d}_{i=1}^{k}, \tilde{d}'^{k}_{i=1}, d_{j=k+1}^{n}\right) \in NS(s) \iff \left(\left(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}\right) \in NS_o(s) \wedge \left(\tilde{d}'^{k}_{i=1}, d_{j=k+1}^{n}\right) \in NS_m(s) \right)\right]$$

Let's now introduce two predicates that permits to discover how a state in the merging model represents the states of the original and mutated models.

Definition 8.5 (*Both* and *Either* predicates). *Let*

$$\begin{array}{lcl} Both(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}) & \triangleq & \bigwedge_{i=1}^{k} \left(\tilde{d}_i = \tilde{v}_i \wedge \tilde{d}_i = \tilde{v}'_i\right) \wedge \bigwedge_{j=k+1}^{n} d_j = v_j \\ Either(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}) & \triangleq & \left(\bigwedge_{i=1}^{k} \tilde{d}_i = \tilde{v}_i \vee \bigwedge_{i=1}^{k} \tilde{d}_i = \tilde{v}'_i\right) \wedge \bigwedge_{j=k+1}^{n} d_j = v_j \end{array}$$

be two predicates such that, given a n-upla of values $d = (\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n})$ (that represents a state), $Both(d)$ means that both models M_o and M_m are in the same state d, while $Either(d)$ means that at least one model is in state d.

8.5.3 Equivalence checking through CTL properties

In this section we will see how the properties described in Theorem 8.1 for ensuring the equivalence of two Kripke structures can be checked through some CTL properties for checking the equivalence of two NuSMV models. Section 8.5.3.1 redefines, for NuSMV models, some predicates previously introduced for Kripke structures. Then, Section 8.5.3.2 describes how to prove properties 8.2 and 8.3 on the initial states, and Section 8.5.3.3 how to prove property 8.4 on the transition relation. Finally, Section 8.5.3.4 presents an example of the application of the technique.

8.5.3.1 Equivalence of the projections and mirror state

Since a state in a NuSMV model is identified by the values of its variables (see Def. 3.10), the predicates for the equivalence of the projections (see Def. 8.1) can be written using the interpretation of the variables in the state, instead of the labels. Let s and s' be states of S_e.

$$le(s) \triangleq \forall v \in MV \;\; [\llbracket v \rrbracket_s = \llbracket mut(v) \rrbracket_s] \triangleq \bigwedge_{i=1}^{k} \llbracket \tilde{v}_i \rrbracket_s = \llbracket \tilde{v}'_i \rrbracket_s \tag{8.5}$$

$$le_o(s,s') \triangleq \forall v \in (MV \cup DV) \;\; [\llbracket v \rrbracket_s = \llbracket v \rrbracket_{s'}] \triangleq \bigwedge_{i=1}^{k} \llbracket \tilde{v}_i \rrbracket_s = \llbracket \tilde{v}_i \rrbracket_{s'} \wedge \bigwedge_{j=k+1}^{n} \llbracket v_j \rrbracket_s = \llbracket v_j \rrbracket_{s'} \tag{8.6}$$

$$le_m(s,s') \triangleq \forall v \in \left(MV' \cup DV\right) \;\; [\llbracket v \rrbracket_s = \llbracket v \rrbracket_{s'}] \triangleq \bigwedge_{i=1}^{k} \llbracket \tilde{v}'_i \rrbracket_s = \llbracket \tilde{v}'_i \rrbracket_{s'} \wedge \bigwedge_{j=k+1}^{n} \llbracket v_j \rrbracket_s = \llbracket v_j \rrbracket_{s'} \tag{8.7}$$

Applying formulae 8.5 and 8.6, the formula $le(s') \wedge le_o(s,s')$ can be written in the following way:

$$\begin{aligned} le(s') \wedge le_o(s,s') &\triangleq \bigwedge_{i=1}^{k} \llbracket \tilde{v}_i \rrbracket_{s'} = \llbracket \tilde{v}'_i \rrbracket_{s'} \wedge \bigwedge_{i=1}^{k} \llbracket \tilde{v}_i \rrbracket_s = \llbracket \tilde{v}_i \rrbracket_{s'} \wedge \bigwedge_{j=k+1}^{n} \llbracket v_j \rrbracket_s = \llbracket v_j \rrbracket_{s'} \\ &\triangleq \bigwedge_{i=1}^{k} (\llbracket \tilde{v}_i \rrbracket_s = \llbracket \tilde{v}_i \rrbracket_{s'} \wedge \llbracket \tilde{v}_i \rrbracket_s = \llbracket \tilde{v}'_i \rrbracket_{s'}) \wedge \bigwedge_{j=k+1}^{n} \llbracket v_j \rrbracket_s = \llbracket v_j \rrbracket_{s'} \end{aligned} \tag{8.8}$$

Applying formulae 8.5 and 8.7, the formula $le(s') \wedge le_m(s,s')$ can be written in the following way:

$$\begin{aligned} le(s') \wedge le_m(s,s') &\triangleq \bigwedge_{i=1}^{k} \llbracket \tilde{v}_i \rrbracket_{s'} = \llbracket \tilde{v}'_i \rrbracket_{s'} \wedge \bigwedge_{i=1}^{k} \llbracket \tilde{v}'_i \rrbracket_s = \llbracket \tilde{v}'_i \rrbracket_{s'} \wedge \bigwedge_{j=k+1}^{n} \llbracket v_j \rrbracket_s = \llbracket v_j \rrbracket_{s'} \\ &\triangleq \bigwedge_{i=1}^{k} (\llbracket \tilde{v}'_i \rrbracket_s = \llbracket \tilde{v}_i \rrbracket_{s'} \wedge \llbracket \tilde{v}'_i \rrbracket_s = \llbracket \tilde{v}'_i \rrbracket_{s'}) \wedge \bigwedge_{j=k+1}^{n} \llbracket v_j \rrbracket_s = \llbracket v_j \rrbracket_{s'} \end{aligned} \tag{8.9}$$

Finally, the predicate *mirror* (see Def. 8.2) for NuSMV models can be defined using formulae 8.5, 8.8 and 8.9.

$$\begin{aligned} & mirror(s) \triangleq \\ & \bigwedge_{i=1}^{k} \llbracket \tilde{v}_i \rrbracket_s = \llbracket \tilde{v}'_i \rrbracket_s \rightarrow \\ & \forall s' \in next(s) \\ & \left[\begin{array}{l} \exists s'' \in next(s) \left[\bigwedge_{i=1}^{k} (\llbracket \tilde{v}_i \rrbracket_{s'} = \llbracket \tilde{v}_i \rrbracket_{s''} \wedge \llbracket \tilde{v}_i \rrbracket_{s'} = \llbracket \tilde{v}'_i \rrbracket_{s''}) \wedge \bigwedge_{j=k+1}^{n} \llbracket v_j \rrbracket_{s'} = \llbracket v_j \rrbracket_{s''} \right] \wedge \\ \exists s''' \in next(s) \left[\bigwedge_{i=1}^{k} (\llbracket \tilde{v}'_i \rrbracket_{s'} = \llbracket \tilde{v}_i \rrbracket_{s'''} \wedge \llbracket \tilde{v}'_i \rrbracket_{s'} = \llbracket \tilde{v}'_i \rrbracket_{s'''}) \wedge \bigwedge_{j=k+1}^{n} \llbracket v_j \rrbracket_{s'} = \llbracket v_j \rrbracket_{s'''} \right] \end{array} \right] \end{aligned} \tag{8.10}$$

8.5.3.2 Equivalence of the initial states

First condition on the initial states Using formula 8.8, formula 8.2 becomes

$$\forall s \in S_e^0, \; \exists s' \in S_e^0 \left[\bigwedge_{i=1}^{k} \left(\llbracket \tilde{v}_i \rrbracket_s = \llbracket \tilde{v}_i \rrbracket_{s'} \wedge \llbracket \tilde{v}_i \rrbracket_s = \llbracket \tilde{v}'_i \rrbracket_{s'}\right) \wedge \bigwedge_{j=k+1}^{n} \llbracket v_j \rrbracket_s = \llbracket v_j \rrbracket_{s'} \right]$$

Substituting the universal quantification over the initial states with the quantification over the values of the variables in the initial states (i.e., IS in Def. 8.3), we obtain

$$\forall(\tilde{d}_{i=1}^{k}, \tilde{d}'^{k}_{i=1}, d_{j=k+1}^{n}) \in IS,\ \exists s' \in S_e^0 \left[\bigwedge_{i=1}^{k} \left(\tilde{d}_i = [\![\tilde{v}_i]\!]_{s'} \wedge \tilde{d}_i = [\![\tilde{v}'_i]\!]_{s'} \right) \wedge \bigwedge_{j=k+1}^{n} d_j = [\![v_j]\!]_{s'} \right]$$

Note that the interpretations of the variables in state s have been replaced with the actual values of the variables in the state.

We can further simply the formula, observing that the values of the variables in MV' (i.e., $\tilde{d}'^{k}_{i=1}$) are not used in the propositional formula (matrix) of the existentially quantified subformula: so it is possible to quantify over IS_o (see Def. 8.3). Moreover, the matrix of the existentially quantified subformula can be replaced with the predicate *Both* (see Def. 8.5) interpreted in state s'. The obtained formula is

$$\forall(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}) \in IS_o,\ \exists s' \in S_e^0\ [\![Both(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n})]\!]_{s'} \tag{8.11}$$

Second condition on the initial states Using formula 8.9, formula 8.3 becomes

$$\forall s \in S_e^0,\ \exists s'' \in S_e^0 \left[\bigwedge_{i=1}^{k} \left([\![\tilde{v}'_i]\!]_s = [\![\tilde{v}_i]\!]_{s''} \wedge [\![\tilde{v}'_i]\!]_s = [\![\tilde{v}'_i]\!]_{s''} \right) \wedge \bigwedge_{j=k+1}^{n} [\![v_j]\!]_s = [\![v_j]\!]_{s''} \right]$$

The quantification over the initial states can be substituted with the quantification over the values of the variables in the initial states, i.e.,

$$\forall(\tilde{d}_{i=1}^{k}, \tilde{d}'^{k}_{i=1}, d_{j=k+1}^{n}) \in IS,\ \exists s'' \in S_e^0 \left[\bigwedge_{i=1}^{k} \left(\tilde{d}'_i = [\![\tilde{v}_i]\!]_{s''} \wedge \tilde{d}'_i = [\![\tilde{v}'_i]\!]_{s''} \right) \wedge \bigwedge_{j=k+1}^{n} d_j = [\![v_j]\!]_{s''} \right]$$

The formula can be further simplified, observing that the values of the variables in MV (i.e., $\tilde{d}_{i=1}^{k}$) are not used in the matrix of the existentially quantified subformula: so, it is possible to quantify over IS_m (see Def. 8.3). Moreover, the matrix of the existentially quantified subformula can be replaced with the predicate *Both* interpreted in state s''. The obtained formula is

$$\forall(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}) \in IS_m,\ \exists s'' \in S_e^0\ [\![Both(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n})]\!]_{s''} \tag{8.12}$$

Unique formula for checking formulae 8.11 and 8.12 In order to use a unique formula for checking formulae 8.11 and 8.12 we must introduce the following theorem.

Theorem 8.2. *Being A, B and C three domains such that $A \cup B \subseteq C$, it holds that*

$$\forall x \in A\ \ [f(x)] \wedge \forall y \in B\ \ [f(y)]\ \equiv\ \forall z \in C\ \ [(z \in A \vee z \in B) \rightarrow f(z)]$$

The matrices of the universal quantified formulae 8.11 and 8.12 are the same. So, it is possible to prove both properties, using the following formula[2]

$$\forall(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j) \left[\begin{array}{l} \exists s \in S_e^0 \left[\left(\bigwedge_{i=1}^{k} [\![\tilde{v}_i]\!]_s = \tilde{d}_i \vee \bigwedge_{i=1}^{k} [\![\tilde{v}'_i]\!]_s = \tilde{d}_i \right) \wedge \bigwedge_{j=k+1}^{n} [\![v_j]\!]_s = d_j \right] \rightarrow \\ \exists s' \in S_e^0\ [\![Both(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n})]\!]_{s'} \end{array} \right]$$

We can rewrite the formula using the predicate *Either* in the antecedent of the implication

$$\forall \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j \right) \left[\begin{array}{c} \exists s \in S_e^0\ [\![Either(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n})]\!]_s \rightarrow \\ \exists s' \in S_e^0\ [\![Both(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n})]\!]_{s'} \end{array} \right] \tag{8.13}$$

[2]In our case $IS_o \subseteq \left(\times_{i=1}^{k} \tilde{D}_i \times \times_{j=k+1}^{n} D_j \right)$ and $IS_m \subseteq \left(\times_{i=1}^{k} \tilde{D}_i \times \times_{j=k+1}^{n} D_j \right)$. So, we can take as C the domain $\times_{i=1}^{k} \tilde{D}_i \times \times_{j=k+1}^{n} D_j$.

Checking equivalence of initial states using CTL properties As we have already seen, in NuSMV a CTL property φ is true iff it is true starting from each initial state, i.e.,

$$M \models \varphi \qquad \text{iff} \qquad \forall s_0 \in S^0 \ (M, s_0) \models \varphi$$

So, if we want to know if a property is true in *at least* an initial state, we must check $\neg\varphi$; if $M \not\models \neg\varphi$, it means that there exists an initial state in which φ is true, i.e.,

$$M \not\models \neg\varphi \qquad \text{iff} \qquad \exists s_0 \in S^0 \ (M, s_0) \models \varphi$$

So, in order to check property 8.13, for all tuples of values $(\tilde{d}^k_{i=1} \in \tilde{D}_i, d^n_{j=k+1} \in D_j)$, we have to:

1) check the CTL property
$$\neg \mathit{Either}\left(\tilde{d}^k_{i=1}, d^n_{j=k+1}\right) \tag{8.14}$$

2) if property 8.14 is false, also check that the CTL property
$$\neg \mathit{Both}\left(\tilde{d}^k_{i=1}, d^n_{j=k+1}\right) \tag{8.15}$$

is false.

8.5.3.3 Equivalence of the transition relations

Applying the definition of the predicate *mirror* (see Formula 8.10), formula 8.4 becomes

$$\begin{array}{l}
\forall s \in \mathit{reach}(M_e) \\
\left[\begin{array}{l}
\bigwedge\limits_{i=1}^{k} [\![\tilde{v}_i]\!]_s = [\![\tilde{v}'_i]\!]_s \rightarrow \\
\forall s' \in \mathit{next}(s) \\
\left[\begin{array}{l}
\exists s'' \in \mathit{next}(s) \left[\bigwedge\limits_{i=1}^{k} ([\![\tilde{v}_i]\!]_{s'} = [\![\tilde{v}_i]\!]_{s''} \wedge [\![\tilde{v}_i]\!]_{s'} = [\![\tilde{v}'_i]\!]_{s''}) \wedge \bigwedge\limits_{j=k+1}^{n} [\![v_j]\!]_{s'} = [\![v_j]\!]_{s''}\right] \wedge \\
\exists s''' \in \mathit{next}(s) \left[\bigwedge\limits_{i=1}^{k} ([\![\tilde{v}'_i]\!]_{s'} = [\![\tilde{v}_i]\!]_{s'''} \wedge [\![\tilde{v}'_i]\!]_{s'} = [\![\tilde{v}'_i]\!]_{s'''}) \wedge \bigwedge\limits_{j=k+1}^{n} [\![v_j]\!]_{s'} = [\![v_j]\!]_{s'''}\right]
\end{array}\right]
\end{array}\right]
\end{array}$$

The formula can be simplified, replacing the universal quantification over the next states of s with the universal quantification over the values of the variables in the next states of s (i.e., $NS(s)$ in Def. 8.4), in the following way

$$\forall s \in \mathit{reach}(M_e)
\left[\begin{array}{l}
\bigwedge\limits_{i=1}^{k} [\![\tilde{v}_i]\!]_s = [\![\tilde{v}'_i]\!]_s \rightarrow \\
\forall \left(\tilde{d}^k_{i=1} \in \tilde{D}_i, \tilde{d}'^k_{i=1} \in \tilde{D}_i, d^n_{j=k+1} \in D_j\right) \in NS(s) \\
\left[\begin{array}{l}
\exists s'' \in \mathit{next}(s) \left[\bigwedge\limits_{i=1}^{k} \left(\tilde{d}_i = [\![\tilde{v}_i]\!]_{s''} \wedge \tilde{d}_i = [\![\tilde{v}'_i]\!]_{s''}\right) \wedge \bigwedge\limits_{j=k+1}^{n} d_j = [\![v_j]\!]_{s''}\right] \wedge \\
\exists s''' \in \mathit{next}(s) \left[\bigwedge\limits_{i=1}^{k} \left(\tilde{d}'_i = [\![\tilde{v}_i]\!]_{s'''} \wedge \tilde{d}'_i = [\![\tilde{v}'_i]\!]_{s'''}\right) \wedge \bigwedge\limits_{j=k+1}^{n} d_j = [\![v_j]\!]_{s'''}\right]
\end{array}\right]
\end{array}\right]$$

In the formula, in the first existentially quantified subformula, the values of the variables MV' (i.e., $\tilde{d}'^k_{i=1}$) are never used, and, in the second existentially quantified subformula, the values of the variables MV (i.e., $\tilde{d}^k_{i=1}$) are never used. So, we can rewrite the formula, splitting the universal quantification over $NS(s)$ in two universal quantifications over $NS_o(s)$ and $NS_m(s)$ (see Def.

8.4). Moreover the matrices of the existentially quantified subformulae can be replaced by the predicate *Both* interpreted in the quantified state. This is the obtained formula:

$$\forall s \in reach(M_e) \left[\bigwedge_{i=1}^{k} [\![\tilde{v}_i]\!]_s = [\![\tilde{v}'_i]\!]_s \rightarrow \left(\begin{array}{l} \forall \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right) \in NS_o(s), \\ \exists s'' \in next(s) \ \left[\!\!\left[Both\left(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}\right)\right]\!\!\right]_{s''} \wedge \\ \forall (\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j) \in NS_m(s), \\ \exists s''' \in next(s) \ \left[\!\!\left[Both\left(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}\right)\right]\!\!\right]_{s'''} \end{array} \right) \right]$$

In the formula, the matrices of the two universally quantified subformulae over $NS_o(s)$ and $NS_m(s)$ are the same. According to Theorem 8.2, the conjunction of the two universally quantified subformulae can be replaced by a single formula universally quantified over the bigger domain $\times_{i=1}^{k} \tilde{D}_i \times \times_{j=k+1}^{n} D_j$[3]. This is the obtained formula:

$$\forall s \in reach(M_e) \left[\bigwedge_{i=1}^{k} [\![\tilde{v}_i]\!]_s = [\![\tilde{v}'_i]\!]_s \rightarrow \left(\begin{array}{l} \forall d = \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right) \\ \left[\begin{array}{l} (d \in NS_o(s) \vee d \in NS_m(s)) \rightarrow \\ \exists s'' \in next(s) \ \left[\!\!\left[Both\left(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}\right)\right]\!\!\right]_{s''} \end{array} \right] \end{array} \right) \right]$$

The formula can be rewritten, transforming the antecedent of the rightmost implication, in the following way

$$\forall s \in reach(M_e) \left[\begin{array}{l} \bigwedge_{i=1}^{k} [\![\tilde{v}_i]\!]_s = [\![\tilde{v}'_i]\!]_s \rightarrow \\ \forall \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right) \\ \left[\begin{array}{l} \exists s' \in next(s) \ \left[\left(\bigwedge_{i=1}^{k} \tilde{d}_i = [\![\tilde{v}_i]\!]_{s'} \vee \bigwedge_{i=1}^{k} \tilde{d}_i = [\![\tilde{v}'_i]\!]_{s'} \right) \wedge \bigwedge_{j=k+1}^{n} d_j = [\![v_j]\!]_{s'} \right] \rightarrow \\ \exists s'' \in next(s) \ \left[\!\!\left[Both\left(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}\right)\right]\!\!\right]_{s''} \end{array} \right] \end{array} \right]$$

We rewrite the formula, extracting the inner universal quantifier, and replacing the matrix of the first existential quantifier with the predicate *Either* (see Def. 8.5).

$$\begin{array}{l} \forall \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right), \\ \forall s \in reach(M_e) \end{array} \left[\bigwedge_{i=1}^{k} [\![\tilde{v}_i]\!]_s = [\![\tilde{v}'_i]\!]_s \rightarrow \left[\begin{array}{l} \exists s' \in next(s) \ [\![Either(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n})]\!]_{s'} \rightarrow \\ \exists s'' \in next(s) \ [\![Both(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n})]\!]_{s''} \end{array} \right] \right]$$

Finally the two implications can be simplified in the following way[4]

$$\begin{array}{l} \forall \left(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j\right), \ \forall s \in reach(M_e) \\ \left[\begin{array}{l} \left(\bigwedge_{i=1}^{k} [\![\tilde{v}_i]\!]_s = [\![\tilde{v}'_i]\!]_s \wedge \exists s' \in next(s) \ [\![Either(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n})]\!]_{s'} \right) \rightarrow \\ \exists s'' \in next(s) \ [\![Both(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n})]\!]_{s''} \end{array} \right] \end{array} \tag{8.16}$$

[3]Note that, for any $s \in S_e$, $\times_{i=1}^{k} \tilde{D}_i \times \times_{j=k+1}^{n} D_j \supseteq NS_o(s) \cup NS_m(s)$.

[4]$P \rightarrow (Q \rightarrow R) \equiv (P \wedge Q) \rightarrow R$

Checking equivalence of the transition relation using CTL properties In NuSMV, checking property 8.16 means checking that the following formula

$$\mathtt{AG}\left(\left(\bigwedge_{i=1}^{k} \tilde{v}_i = \tilde{v}_i' \wedge \mathtt{EX}\left(\mathit{Either}\left(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}\right)\right)\right) \rightarrow \mathtt{EX}\left(\mathit{Both}\left(\tilde{d}_{i=1}^{k}, d_{j=k+1}^{n}\right)\right)\right) \tag{8.17}$$

holds in M_e, for each tuple of values $(\tilde{d}_{i=1}^{k} \in \tilde{D}_i, d_{j=k+1}^{n} \in D_j)$.

Formula 8.17 has been obtained from formula 8.16 simply applying the semantics of the CTL operators AG and EX:

- $M \models \mathtt{AG}(\varphi)$ iff $\forall s \in \mathit{reach}(M)\ [(M, s) \models \varphi]$
- $M, s \models \mathtt{EX}(\varphi)$ iff $\exists s' \in \mathit{next}(s)\ [(M, s') \models \varphi]$

8.5.3.4 Example of equivalence checking

Let's see the application of the CTL properties shown in Formulae 8.14, 8.15 and 8.17, to check the equivalence of models shown in Codes 8.1 and 8.2.

In the following, we report some of the CTL properties derived from Formulae 8.14, 8.15. Some of them must be checked against the merging model (Code 8.4) in order to prove the equivalence of the two models in their initial states.

```
CTLSPEC NAME isNotInitState_1 := !((amPm = AM | amPmMut = AM) & hour = 0)
CTLSPEC NAME notEqInitState_1 := !((amPm = AM & amPmMut = AM) & hour = 0)
CTLSPEC NAME isNotInitState_2 := !((amPm = AM | amPmMut = AM) & hour = 1)
CTLSPEC NAME notEqInitState_2 := !((amPm = AM & amPmMut = AM) & hour = 1)
 ...
CTLSPEC NAME isNotInitState_25 := !((amPm = PM | amPmMut = PM) & hour = 0)
CTLSPEC NAME notEqInitState_25 := !((amPm = PM & amPmMut = PM) & hour = 0)
 ...
CTLSPEC NAME isNotInitState_48 := !((amPm = PM | amPmMut = PM) & hour = 23)
CTLSPEC NAME notEqInitState_48 := !((amPm = PM & amPmMut = PM) & hour = 23)
```

We must check that, if a CTL property *isNotInitState_i* is false, then also the CTL property *notEqInitState_i* is false. In the example, we verified that *isNotInitState_1* and *notEqInitState_1* are false (i.e., there is an initial state in which *hour* is 0 and both *amPm* and *amPmMut* are *AM*), and all the properties *isNotInitState_i*, with $i = 2, \ldots, 48$, are true: so the two models are equivalent in the unique initial state. Totally, we had to check 49 over 96 properties.

We now report some of the CTL properties derived from Formula 8.17. All the properties must be checked against the merging model in order to prove the equivalence of the transition relations of the two models; if the two models are not equivalent, it could be that not all the properties must be checked.

```
CTLSPEC NAME transRelOk_1 :=
          AG( (amPm = amPmMut & EX((amPm = AM | amPmMut = AM) & hour = 0)) −>
              EX((amPm = AM & amPmMut = AM) & hour = 0) )
 ...
CTLSPEC NAME transRelOk_12 :=
          AG( (amPm = amPmMut & EX((amPm = AM | amPmMut = AM) & hour = 11)) −>
              EX((amPm = AM & amPmMut = AM) & hour = 11) )
 ...
CTLSPEC NAME transRelOk_36 :=
          AG( (amPm = amPmMut & EX((amPm = PM | amPmMut = PM) & hour = 11)) −>
              EX((amPm = PM & amPmMut = PM) & hour = 11) )
 ...
CTLSPEC NAME transRelOk_48 :=
          AG( (amPm = amPmMut & EX((amPm = PM | amPmMut = PM) & hour = 23)) −>
              EX((amPm = PM & amPmMut = PM) & hour = 23) )
```

We must check that all the CTL properties *transRelOk_i*, with $i = 1, \dots, 48$ are true. As soon as we find a property false, we can stop checking since we will have found that the two models are not equivalent. In the example, we only checked the first 12 properties, since we found that *transRelOk_12* is false[5].

So, the original model (Code 8.1) and its mutant (Code 8.2) are not equivalent.

8.6 Mutation and model review

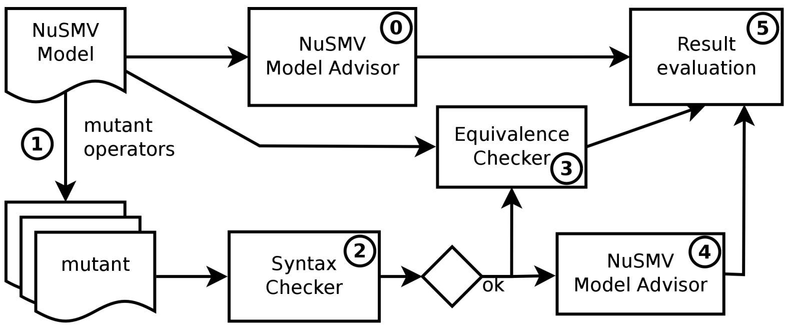


Figure 8.1: Assessing the fault detection capability of a model review technique using mutation analysis

We have devised the following framework with the aim of assessing the fault detection capability of the NuSMV Model Advisor. The process is depicted in Fig. 8.1. A NuSMV model is taken as input by the evaluation process. The mutation operators presented in Section 8.2 are applied in order to obtain a large number of mutations of the original NuSMV model (step 1). Each mutant is then parsed by the original NuSMV parser in order to detect those mutations which result in syntax errors (step 2). These mutations represent mistakes that can be easily detected by syntax, type, and dependency checking performed by NuSMV itself. Every mutant that survives is analysed by the *equivalence checker* in order to establish if it is equivalent to the original specification (step 3), and it is checked by the NuSMV model advisor in order to assess the violations of meta-properties in the mutant (step 4). The original NuSMV specification must be previously reviewed (step 0) and the results obtained over the mutant are compared with the original review in order to establish if the mutant is either *killed* or not by the model advisor (step 5). The decision of killing a mutant is taken as follows.

Let

$$MpV_i : NuSMVmodels \rightarrow \mathbb{N}_0$$

be the function that returns the number of violations of meta-property $\mathrm{MP_N}i$ for any NuSMV model (the set *NuSMVspecs* contains both the original and the mutated specifications).

Definition 8.6. *We say that the model advisor kills a mutant (let's call it* mut*) if some meta-property is violated more times than that of the original specification (see Formula 8.1), i.e., formally*

$$\exists \mathrm{MP_N}i : MpV_i(mut) > MpV_i(orig(mut))$$

Depending on the number of violations of the original specification, we can distinguish two scenarios:

1. **Original models without meta-properties violations**. In case the original model does not violate any meta-property (it represents a completely corrected specification, i.e., $\forall \mathrm{MP_N}i : MpV_i(orig(mut)) = 0$), the mutant is killed if $MpV_i(mut) > 0$ for any $\mathrm{MP_N}i$.

[5]Note that also property *transRelOk_36*, if checked, would have been violated.

2. **Original models with meta-properties violations**: In this case original models violate some meta-properties. This is the most common case, since a model with some violations could still be acceptable (once that the designer has checked that those violations are not faults in the model). This is particularly true for meta-properties that refer to the style in which the specification is written (like MP_N3 or MP_N4). In order to kill a mutant, an increase of the number of violations of *at least a* meta-property suffices. Note that the total number of meta-property violations may decrease due to some mutations: we do not only consider the total number of meta-property violations, because we are interested in analysing the contribution of every single meta-property.

8.6.1 Classification of the results

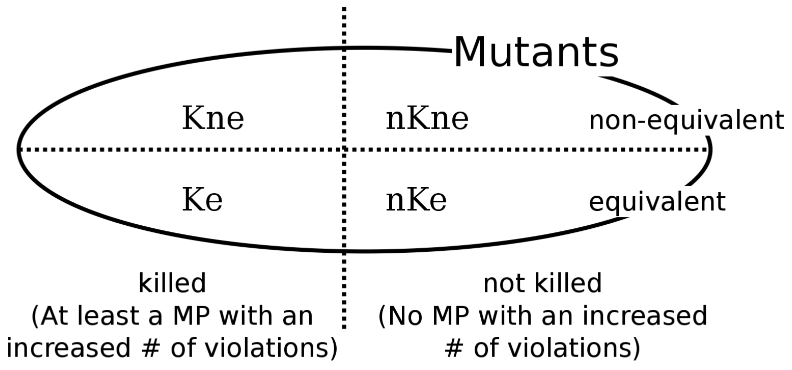

Mutants
Kne
nKne
non-equivalent
Ke
nKe
equivalent
killed
(At least a MP with an increased # of violations)
not killed
(No MP with an increased # of violations)

Figure 8.2: Classification of the results

The evaluation of the results takes into account the problem of mutant killing and equivalence. We can classify four cases, as depicted in Fig. 8.2. The set of mutants representing all the specifications obtained by mutation, can be divided in two by considering equivalence and by two again by considering if a mutant is killed or not. This results into the following four subsets.

Kne: A non-equivalent mutant (from now on *neq-mutant*) is killed by the model advisor. This means that the mutation, representing a real fault, causes a violation of some meta-property (or an increase of the number of violated meta-properties) and that the model advisor is capable of finding these faults. In brief: the model advisor finds a real fault.

Ke: An equivalent mutant (from now on *eq-mutant*) is killed by the model advisor. This means that the mutation does not change the behaviour of the machine, nevertheless it changes the specification in a way that a meta-property is (more) violated. This represents a *false positive*: the model advisor marks as fault a mutation in the structure which could be classified as "stylistic" defect.

nKe: An eq-mutant is not killed by the model advisor. The mutation does not change the behaviour of the machine neither modifies the structure of the specification in a way that some meta-property is violated.

nKne: A neq-mutant is not killed by the model advisor. This means that a real fault passes undetected and represents a *false negative.*

While *Kne* and *nKe* represent the correct outcome of the analysis, *Ke* and *nKne* represent mistakes but with different meanings. False positives (*Ke*) prompt the user to modify a specification that is fault-free but that, nevertheless, may have other problems like readability. On the contrary, false negative cases (*nKne*) are a clear sign of weakness in the model review we propose, since the model review is not enough powerful to discover such faults. This case would require to introduce new meta-properties (if possible) or the developer to perform other types of analysis.

Fault detection capability

If we just consider the fault detection capability, only neq-mutants must be taken into account, since they alone represent real faults. In this case the model advisor behaves correctly when it kills neq-mutants (Kne), while it is mistaken only when it does not kill neq-mutants (nKne). To increase the fault detection capability, we should increase the ratio between Kne and the number of neq-mutants.

However, although a high number of false positives (Ke) does not diminish the fault detection capability, it may weaken the quality of the analysis since the designer may decide to ignore the results all together if the number of requested modifications is too high.

8.7 Experiments

The experiments have been executed on a Linux machine, Intel(R) Core(TM) i7 CPU, 4 GB RAM; the model advisor tool is part of the nusmv-tools framework [138]; the NuSMV version used is 2.5.4.

We have collected 104 NuSMV specifications from different sources:

- examples contained in the NuSMV distribution (9);
- specifications sent to the NuSMV mailing list (3);
- specifications found on the Internet (research works, teaching material, etc.) (39);
- specifications we developed by ourselves for research and teaching purposes (25);
- NuSMV models obtained from the translation of ASMs into NuSMV models through the tool AsmetaSMV (see Section 9.2) (28).

We evaluated the size of the specifications in terms of number of BDD variables[6]. The number of BDD variables is a good indicator of the state space size, since it depends on the number of variables (and not definitions) and the sizes of their domains. The average number is 20.44, the minimum is 2 and the maximum 156.

Moreover we have also considered the number of BDD nodes allocated. The average number is 10650, the minimum is 19 and the maximum 380808.

We have performed the experiments dividing the specifications in the following two sets:

- *NoViolations* containing 51 specifications that do not violate any meta-property;
- *Violations* containing 53 specifications that violate at least one meta-property.

We did not make any selection on the specification kind: we tried to build the sets as much heterogeneous as possible, since our aim is to assess the model advisor fault detection capability not just on a particular kind of specification, but on any possible specification. We know that there are some specifications that are not targeted by the model advisor: indeed, the meta-properties of the model advisor have been originally designed for checking NuSMV specifications that describe the transition relation in an *operational* way (i.e., using the ASSIGN section) and not those that do it in a *declarative* way (i.e., using the INIT and TRANS sections). So, the mutants of specifications belonging to the latter type would probably generate a high number of

[6]Note that the number of BDD variables is the double of value one would expect and that can be discovered, for example, using in NuSMV the command *write_order* that print the order of the BDD variable. This is due to the fact that each BDD variable has two copies, one for the current state, and another for the next state. Also the authors of NuSMV reports the number of BDD variables in this way [49].

false negative results (*nKne*). Among all the 104 specifications, 68 are given in an operational style.

The mutated specifications have been obtained applying 8 mutation operators among those described in Section 8.2: ER, LOR, MOR, ROR, MB, SB, SA0, and SA1. We base our experiments on two classical hypotheses: the *competent programmer* and the *coupling effect.* We decided to apply only one mutation operator at the time (first order mutants), even if our approach supports also higher order mutants (homs) [105], because the number of homs can be very high and because we believe that the coupling effect is generally valid for NuSMV specifications as it is for programs [140].

8.7.1 Overall results analysis

Generated mutants

We generated 26978 mutants; 11% (2946) are rejected by the NuSMV parser, whereas the remaining 24032 are syntactically correct NuSMV specifications. The following table reports, for each mutation class, the number of its mutants and the percentage of them rejected by the NuSMV parser.

Mutation	ER	LOR	MOR	ROR	MB	SB	SA0	SA1
Number of mutants	3037	8540	400	5579	716	1498	3642	3566
Rejected by parser (%)	11	5	38	23	38	0	9	3

A lot of MB mutants have been rejected by the parser: indeed, most of the mutants in which the default condition of a case expression has been removed are likely detected by the parser since the case conditions become not exhaustive. The SB mutants, instead, can not be detected by the parser: indeed, changing the order of the conditions in a case expression does not modify their exhaustiveness.

From now on we only consider mutants that survived the NuSMV parser.

The size of a mutant in terms of number of BDD variables is the same as that of its original specification, since we do not mutate variables declarations, i.e., we do not add/remove variables or values from the variables domains. However, if we consider the average size of a mutant, we discover that the average number of BDD variables is 34.05: such value is greater than the average value of the original specifications seen previously (20.44). This is due to the fact that *bigger* specifications in terms of number of BDD variables, on average, generate more mutants because, in general, they have more instructions that can be affected by a mutation.

The number of allocated BDD nodes of a mutant, instead, can be different from that of the original specification because a mutation can change the behaviour of a model and so also the structure of the underlying BDD. In order to evaluate the difference between a mutant *mut* and its original specification (see Formula 8.1), we use the percentage change, i.e.,

$$percChange(mut) = \frac{\text{BDD\_NA}(mut) - \text{BDD\_NA}(\text{orig}(mut))}{\text{BDD\_NA}(\text{orig}(mut))} * 100 \tag{8.18}$$

where $BDD\_NA : NuSMVspecs \rightarrow \mathbb{N}_0$ is a function that, given a NuSMV specification, retrieves the number of BDD nodes allocated.

Given a set of mutants *Muts*, we identify with

$$\{percChange(mut)\}_{mut \in Muts} \tag{8.19}$$

the set of all the percentage changes of the mutants in *Muts*.

We computed the average and the standard deviation of the set described by Formula 8.19, considering as *Muts* all the mutants, or only the mutants produced with a mutation operator. Results are reported in Table 8.1.

Muts	All	ER	LOR	MOR	ROR	MB	SB	SA0	SA1
AVG of formula 8.19	-0.18	-0.04	1.03	-0.32	1.90	-3.91	-3.37	-3.23	-0.91
STD of formula 8.19	7.35	3.92	6.73	6.47	7.54	7.09	13.27	6.69	5.82

Table 8.1: Average and standard deviation of the percentage change between the number of allocated BDD nodes in a mutant and that of the original specification

We discovered that, on average, mutation operators can increase or decrease the number of allocated BDD nodes with almost equal probability since the percentage change computed over all the mutants is -0.18%, and the data are normally distributed.

However, we discovered that there are some differences among the mutants generated with different kinds of mutation operators. Almost all the operators, except LOR and ROR, on average diminish the number of BDD nodes allocated. This is due to the fact that, most of the times, the application of these operators causes the removal of some behaviours of the model, making the BDD more simple. A MB mutant, in which a branch in a case expression is removed, would probably be more simple than the original model, since, most likely, a transition has been removed. In LOR and ROR mutants, instead, the logical/relational operators are replaced: such replacements do not necessarily simplify the transition relation.

It is interesting to notice that the operator SA0 simplifies the BDD much more than SA1. This is due to the fact that, in the considered specifications, there are a lot of AND-expressions and setting an operand of an AND-expression to FALSE means to stuck at FALSE all the AND-expression, so simplifying the BDD. Stucking at TRUE an operand of and AND-expression, instead, could change the behaviour of the model not so deeply[7].

If we consider the standard deviation of the percentage change, we notice that some mutation operators sometimes can modify the behaviour of the original specification (increasing or decreasing the number of allocated BDD nodes) more than others. The operator SB, for example, can deeply change the behaviour of a model. In fact, swapping the branches in a case expression in which the conditions are not mutually exclusive, could radically change the behaviour of the model: if, for example, the default condition is swapped with the first branch, the corresponding variable is forced to be always updated to the value of the right_expression of the default condition.

Execution times

The total time taken to execute the NuSMV model advisor over the original specifications is 152 seconds, while executing the NuSMV model advisor over the mutated specifications took about 20 hours. The total time taken to check the equivalence between the original and the mutated specifications is almost 7 hours. Although the process for checking the equivalence is computationally expensive in the worst case (when the mutant is equivalent to the original specification), on average it takes 1.04 seconds for specification, because, when the models are not equivalent, not all the CTL properties derived from Formulae 8.14, 8.15 and 8.17 must be checked.

Number of eq-mutants

We are now interested in knowing what is the percentage of eq-mutants. In [85] they found that 40% (8/20) of mutants of a Java code were equivalent. In our experiments we found that 27% (6396/24032) of mutants are equivalent. Both results confirm that eq-mutants are a real problem, since their number is not irrelevant.

[7]We remind to Section 8.7.2.1 for a deeper analysis of the relation between the logical operators of a specification and the application of the mutation operators SA0 and SA1.

	neq-mutants	**eq-mutants**
not killed	45%	53%
killed	55%	47%

Table 8.2: (Not) Equivalent (not) killed mutants

Influence of equivalence on the killability

We observed that the model advisor kills neq-mutants more easily, as shown in Table 8.2.

8.7.2 Quality evaluation

Now we want to make a more general evaluation, assessing *sensitivity*, *precision*, and *accuracy* of our approach. According to the ISO standard [104], for binary classification we can assess the quality factors of our methodology following the schema shown in Table 8.3.

Analysis outcome	Mutant		
	not equivalent	equivalent	
Positive (killed)	true positive *Kne*	false positive *Ke*	→ **Precision**
Negative	false negative *nKne*	true negative *nKe*	
	↓ **Sensitivity**		↘ **Accuracy**

Table 8.3: Quality factors

Note that *sensitivity* considers only neq-mutants, and it measures the actual fault detection capability (since only neq-mutants represent actual faults), whereas *accuracy* measures also the capability of avoiding killing eq-mutants. Finally, *precision* measures the likelihood that a killed mutant is not equivalent.

In the following, we report a list of *hypotheses* we formulated about our methodology. They have been all confirmed by the Kruskal-Wallis one-way analysis of variance [116] that rejected the corresponding null hypotheses. We use the Kruskal-Wallis test because our data are not normally distributed.

8.7.2.1 Sensitivity

In order to discover how good is the model advisor in killing neq-mutants, following the terminology of the theory of classification [104], we introduce *sensitivity* as

$$\text{sensitivity} = \frac{\text{Kne}}{\#\textit{neq-mutants}} = \frac{\text{Kne}}{\text{Kne} + \text{nKne}}$$

Table 8.2 reports the overall sensitivity (55%) measuring the actual fault detection capability of the model advisor.

Specification influence on the sensitivity We want now to discover if some of the characteristics of the original specification influence the killability of its neq-mutants.

We found that the number of killed neq-mutants strongly depends on the specification under analysis. In Fig. 8.3 we show the minimum percentage of killed neq-mutants: for example, for the 74% of specifications, at least the 60% of their neq-mutants are killed. Note that, for 28% of the specifications, 100% of neq-mutants were killed. Can we identify any characteristic of the specifications that influences the sensitivity?

We found that specifications with meta-property violations (set *Violations*) are more sensitive to mutation, that is the model advisor is more sensitive for specifications with some defects. The model advisor kills more easily neq-mutants of specifications in *Violations* than those in

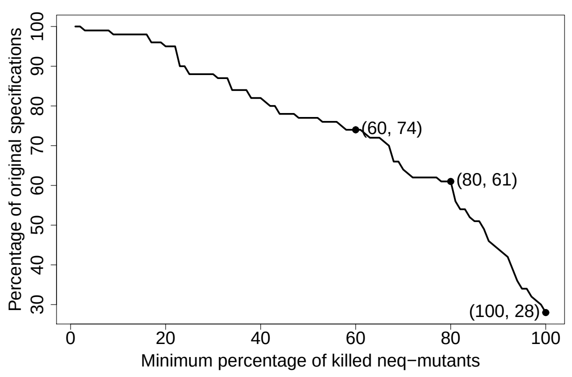


Figure 8.3: Percentage of at least killed neq-mutants vs percentage of specs

$\mathbf{MP_N}$	1	2	3	4	5	6	7	8	9	10
Sensitivity (%)	19	24	19	10	5	23	0	0	26	7
Precision (%)	87	85	66	66	91	89	100	100	100	100
Accuracy (%)	39	41	33	30	30	42	27	27	45	31
nKe/eq-mutants %	92	88	74	86	99	93	100	100	100	100

Table 8.4: Meta-property Sensitivity, Precision, Accuracy, and Not killed eq-mutants

NoViolations. On average, 60% of neq-mutants of specifications in *Violations* are killed, whereas the 49% of neq-mutants of specifications in *NoViolations* are killed.

We found that specifications with temporal properties are more sensitive to mutation. There is a correlation between the fact that a specification has temporal properties and the killability of its neq-mutants; indeed, mutants changing the specification behaviour can be easily killed by meta-properties $\mathrm{MP_N}9$ and $\mathrm{MP_N}10$ checking that each temporal property is, respectively, true and not vacuously true (see Section 5.2.4). We discovered that, on average, 63% of neq-mutants of specifications that have a temporal property are killed, whereas the 41% of neq-mutants of specifications that does not have a temporal property are killed. This result suggests that is a good practice to add temporal properties that redundantly specify what is the desired behaviour of the specification.

We found that specifications in an operational style are more sensitive to mutation. Neq-mutants of specifications whose behaviour is given using ASSIGN constructs, are detected more easily than the average. The sensitivity for these mutants raises from 55% to 77%.

Meta-property sensitivity We want now to discover how much a meta-property is able to kill any kind of neq-mutant.

We found that some meta-properties have greater killing neq-mutants skills (sensitivity). There exists a relation between the number of killed neq-mutants and the meta-property used to kill them. Table 8.4 (row Sensitivity) reports the percentage of neq-mutants that each meta-property kills.

Some meta-properties, like $\mathrm{MP_N}7$ and $\mathrm{MP_N}8$, can not detect any mutation; their aim is to identify models where some *monitored* variables (i.e., variables whose value is unbounded) or *independent* variables (i.e., variables that do not depend on any other variable) are never read: it is difficult to obtain such kind of models with the mutation operators we have tested.
Each single meta-property never kills more than 26% of neq-mutants. As reported in Table 8.2, the model advisor can globally kill more than the half of neq-mutants (55%), which is less than the total sum of all the values. This means that some mutants are killed by more than one meta-property and that the meta-property targets are not disjoint.

Mutation	ER	LOR	MOR	ROR	MB	SB	SA0	SA1
Sensitivity (%)	79	51	83	36	81	39	82	40
Precision (%)	82	74	90	75	79	76	79	69
Accuracy (%)	68	51	84	44	71	61	70	44

Table 8.5: Dependence between the Sensitivity, Precision and Accuracy, and the mutation classes used to generate the mutants

Relation between neq-mutants killability and their mutation classes We want to discover if the neq-mutants of some mutation classes are more easily killed by the model advisor.

We found that some mutation classes generate neq-mutants that have greater killability.

Table 8.5 (row Sensitivity) reports, for each mutation class, the percentage of its neq-mutants that have been killed. The model advisor kills more easily some mutants because their mutation classes often introduce faults in elements of the specification that it checks. There are other mutants, instead, that the model advisor kills very rarely because it is not able to capture most of the faults their mutation classes introduce. It is interesting that killing SA0 mutants is twice easier than killing SA1 mutants; we tried to understand why. SA0 and SA1 replace the operands in logical expressions with, respectively, FALSE and TRUE. We analysed our specifications and we recorded the number of occurrences of logical operators. We found that the & operator occurs 1446 times, whereas the | operator occurs 223 times.

Operator	&	|	$\rightarrow$	*xor*	$\leftrightarrow$	*xnor*
# of occurrences	1446	223	19	20	0	0

If one sets to FALSE an operand of an AND-expression, the expression is forced to be false in all the states; if in the original specification the expression is true in some state (this is highly probable), it is highly probable that the behaviour of the specification is changed and that the model advisor can kill the mutant.
If one sets to TRUE an operand of an AND-expression, instead, the expression is not forced to assume a truth value in all the states: the value of the other operand is still necessary to evaluate the overall expression. So, for the model advisor it is more difficult to kill the mutant, since the mutation could have not changed the behaviour of the specification.

For OR-expressions the reasoning is the opposite: the model advisor kills more easily SA1 mutants rather than SA0 mutants. But, since the number of AND-expressions is more than 6 times the number of OR-expressions (and so the number of their mutants), the number of killed SA0 mutants is higher than the number of killed SA1 mutants.

8.7.2.2 Precision

Following the terminology of the theory of classification [104], we define *precision* as

$$\text{precision} = \frac{\text{Kne}}{\#\textit{killed-mutants}} = \frac{\text{Kne}}{\text{Kne} + \text{Ke}}$$

Precision indicates what is the probability that, once the model advisor kills a mutant, the mutant is not equivalent. The overall precision of the model advisor is 76%.

Meta-property precision We want to know what is the probability that a mutant killed with a given meta-property is not equivalent.

We found that some meta-properties have greater precision. Table 8.4 (row Precision) reports the relation between the precision and the meta-property used. Eight meta-properties are quite precise: when they kill a mutant, it is highly probable that it is not equivalent. Only $\text{MP}_\text{N}3$ and $\text{MP}_\text{N}4$ are not so precise, since they can kill also stylistic defects that do not change the

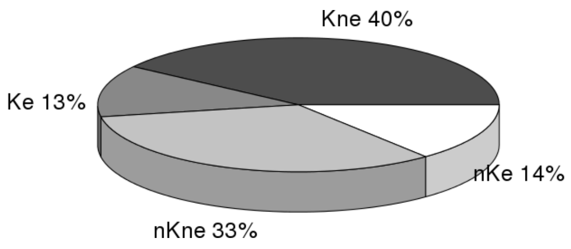


Figure 8.4: Overall results

behaviour of the specification; this fact is confirmed by the ratio between nKe and eq-mutants reported in Table 8.4 (row nKe/eq-mutants) where MP_N3 and MP_N4 have the lowest ratios in not killing eq-mutants.

Relation between the precision of the model advisor and the mutation classes used to generate the mutants We want to discover if mutants of some mutation classes permit to obtain better precision in the model advisor.

We found that some mutation classes generate mutants that are captured by the model advisor with a greater precision. The mutation class used to generate the mutants influences the precision. Table 8.5 (row Precision) reports, for each mutation class, the precision of the model advisor over the mutants of that class. We know that it is difficult to kill the mutants of ROR (Table 8.5 - Sensitivity 36%). Nevertheless, when we kill a mutant of ROR, we know that it is not equivalent with 75% of probability (Table 8.5 - Precision).

8.7.2.3 Accuracy

Now we want to do a more general evaluation; we do not want just to consider how good is the model advisor in killing neq-mutants, but we also want to know how often it does not kill the equivalent ones.

Indeed, as we have seen previously, the model advisor can kill also eq-mutants since some meta-properties can identify stylistic defects that do not change the behaviour of the specification.

Following the terminology of the theory of classification [104], we define *accuracy* as

$$\text{accuracy} = \frac{\text{Kne} + \text{nKe}}{\#mutants} = \frac{\text{Kne} + \text{nKe}}{\text{Kne} + \text{Ke} + \text{nKne} + \text{nKe}}$$

In Fig. 8.4 we show the classification of all the mutants. The value of the overall accuracy of the model advisor is 54% (Kne% + nKe%).

Meta-property accuracy We want now to discover how much a meta-property is able to kill any kind of neq-mutant and not kill the equivalent ones.

We found that some meta-properties have greater accuracy. Table 8.4 (row Accuracy) reports the accuracy of each meta-property. The accuracy of each meta-property is always greater than its sensitivity: this means that each meta-property is good in not killing eq-mutants. This fact is confirmed by Table 8.4 (row nKe/eq-mutants). In general, each meta-property kills very few eq-mutants. However, in Table 8.2 we see that 47% of the eq-mutants are killed: this means that the abilities of killing eq-mutants of the meta-properties are partially disjoint.

Relation between the accuracy of the model advisor and the mutation classes used to generate the mutants We want to discover if the mutants of some mutation classes permit to obtain better accuracy in the model advisor.

We found that some mutation classes generate mutants that are captured by the model advisor with a greater accuracy. Table 8.5 (row Accuracy) reports the accuracy of the model advisor over the mutants of each mutation class. The model advisor obtains greater accuracy than sensitivity when checking the mutants of some mutation classes (e.g., SB), and greater

Mutation	MP_N	Sensitivity (%)	Precision (%)	Accuracy (%)
MOR	MP_N6	48.68	100	68.42
MOR	MP_N5	38.16	100	61.94
SA0	MP_N6	53.12	93.88	61.87
SB	MP_N6	28.57	87.16	60.41
SB	MP_N3	31.12	81.06	60.15
MOR	MP_N2	34.21	100	59.51
SB	MP_N2	33.29	73.94	58.95
MOR	MP_N1	32.89	100	58.7
SB	MP_N1	17.22	87.1	55.34
SB	MP_N9	18.75	77.37	54.61
...			...	
ROR	MP_N8	0	100	26.18
SA1	MP_N4	3.02	35.21	26.08
LOR	MP_N7	0	100	25.85
LOR	MP_N8	0	100	25.85
SA0	MP_N7	0.48	100	24.62
SA0	MP_N8	0.16	100	24.38
ER	MP_N3	16.43	60.33	22.17
SA0	MP_N3	2.55	18.82	17.87
ER	MP_N7	0	100	17.53
ER	MP_N8	0	100	17.53

Table 8.6: Relationship between mutation classes and meta-properties in terms of Accuracy

sensitivity than accuracy when checking the mutants of other mutation classes (e.g., SA0). For instance, accuracy with SB (61%) is definitely greater than sensitivity (39%) since the model advisor kills very rarely SB eq-mutants. Indeed, eq-mutants swapping two case branches could be killed only by meta-properties MP_N2 and MP_N4.

8.7.2.4 Relation between mutation classes and meta-properties

We check the correlation between mutation classes and meta-properties in killing mutants. We want to discover which mutations are targeted with the highest/lowest probability by which meta-properties. We evaluate the relationship between a mutation class *MUT* and a meta-property *MP*i by considering the value of the accuracy obtained by using only *MP*i over the mutants produced with *MUT*. We compute the accuracy because it is the best indicator of the reliability of the proposed approach, since, unlike sensitivity, it also considers the not killed eq-mutants. Table 8.6 reports the couples (mutation class, meta-property) sorted in descending order by the value of the accuracy: we report the first and the last ten couples with their values of sensitivity, precision and accuracy.

As expected, the best/worst couples, in general, are composed by those mutation classes and meta-properties that obtain the best/worst results in Tables 8.5 and 8.4 (Row Accuracy), that show, respectively, the accuracy of the model advisor in capturing the mutants of each mutation class, and the accuracy of each meta-property in capturing all the mutants.

However, there are some exceptions due to the strong correlation that may (not) exist between a mutation class and a meta-property.

For example, MP_N5 does not have a good accuracy over all the mutants (30% in Table 8.4); however, if we only consider the mutants obtained with the mutation class MOR, the accuracy is 61.94%, the second best result in Table 8.6.

Mutants obtained with the mutation class SA0 are captured by the model advisor with a very good accuracy (the second best value in Table 8.5), and meta-property $\mathrm{MP_N3}$ has a decent accuracy over all the mutants (fifth best value in Table 8.4). However, $\mathrm{MP_N3}$ has a very bad accuracy over the mutants of SA0 (third last value in Table 8.6).

8.7.2.5 Overall evaluation

Besides sensitivity, precision, and accuracy, if we also consider the ability of killing stylistic defects, the model advisor behaves correctly in 67% of the cases (Kne + nKe + Ke) as shown in Fig. 8.4.

In conclusion, based on the statistical analysis performed, a user can assess the expected success of finding faults by the use of the model advisor, depending on the characteristics of the specification, on the type of targeted faults, and on the type of violations of meta-properties. A heuristic procedure could, given a specification (with some features) and the list of violated meta-properties, compute the probability that the specification contains each kind of fault we have identified.

Part III

Verification and runtime monitoring

Chapter 9

Model verification

A definition given by Boehm of *system verification* is that it consists of all those techniques that permit to answer the question "Are we building the product right?" [29].

We here only consider *model verification*, i.e., the process of discovering if the formal specification is *correct*. Model verification should be only applied when a designer has enough confidence that her model captures all informal requirements, thus after model validation. The aim of model verification is not to discover if the specification fulfils the intended requirements (the aim of model validation), but if it implements them in the right way. Model verification is the execution of expensive (in terms of execution time) and accurate analyses of the model for, for example, the proof of mathematical theorems or the verification of properties. Two main verification techniques are *theorem proving* and *model checking*.

Using *theorem proving* a user can check mathematical theorems about the model with the help of a program [146], the *theorem prover*, that checks that all the proof steps are correctly derived from the proof system used. A key characteristic of a theorem prover is the degree of automation that it can provide; an *automated theorem prover* can require few user assistance for producing a complete formal proof. Moreover it can help the user in choosing how to continue the proof starting from a given point. Useful features of theorem provers are the possibility of exploring the proof for checking the previous steps and finding alternatives, backtracking, and reusing proofs in other proofs. Since *theorem provers* reason about the *syntactic domain* of a model (or program), they can also reason about infinite state spaces [141]. The main drawback of theorem provers is that they usually *require a great deal of user expertise and effort* [112].

Model checking is a different approach to verification based on the exploration of the model state space to check that some desired properties hold. Model checking is usually described as a "push button" technique, since the verification of the properties is fully automatic, without the need of user intervention. In Section 9.1 we briefly introduce model checking, and in Section 9.2 we propose a technique for model checking ASMs.

9.1 Model checking

Model checking is an automated formal verification technique whose aim is to discover if an abstract description $\mathcal{M}$ of a system satisfy a property φ, i.e.,

$$\mathcal{M}, s \models \varphi$$

where s is a state of $\mathcal{M}$.

The model checking technique consists in the exhaustive exploration of the state space of $\mathcal{M}$ to check if property φ holds. Model checkers, using explicit state enumeration, can handle state spaces of 10^8 to 10^9 states, whereas, using tailored data structures, also state spaces of 10^{20} to 10^{476} states can be checked for particular problems [16].

The model checking process can be divided in three phases [16]:

1. the *modeling* phase:

   (a) the system is modeled using the model checker notation. Since the property verification can be very time-consuming, it is better to be enough confident that the model really reflects the system under verification. This can be achieved by simulating the model and by using a model review technique as, for example, the one presented in Chapter 6 for the NuSMV model checker.
   (b) formal properties are specified using a logic supported by the model checker.

2. the *verification* phase: the model checker, in an automated way, checks the model to discover if it assures the specified properties.

3. the *analysis* phase: the model checker, as result of the verification phase, for each specified property returns a positive or negative response depending on the fact the the property is satisfied or not. In case of negative response, it usually also returns a *counterexample*, in the form of an execution trace of the model, that provides a *witness* of the property failure.

In Section 9.1.1 we see how it is possible to describe the model $\mathcal{M}$, and in Section 9.1.2 what logics are available to specify the required properties. We are not interested here in the algorithms used to solve the model checking problem [16].

9.1.1 Model

The behaviour of the system is usually modeled using a *finite-state automaton*, where the *states* represent the system states and the *transitions* describe how the system changes its state depending on some particular conditions [135].

There are mainly three notations used to describe the model[1]:

- *Kripke structures*;
- *Labeled transition systems*;
- *Kripke transition systems.*

Kripke structures (KSs) have been described in Section 3.1. For model checking purposes they are required to be finite (see Def. 3.2) and total (see Def. 3.3). KSs are particular suitable for describing the properties of the states.

Labeled transition systems (LTSs) [135] differ from Kripke structures since they label transitions with some *actions*, rather than states. LTSs can be defined as follows.

Definition 9.1 (Labeled transition system)**.** *A* labeled transition system *is defined by the 4-tuple* $\langle S, S_0, A, \rightarrow \rangle$*, being*

- S *the set of states;*
- $S_0 \subseteq S$ *the set of initial states;*
- A *the set of* actions*;*
- $\rightarrow \subseteq S \times A \times S$ *the transition relation. The transition* (s, a, s')*, that can be represented as* $s \xrightarrow{a} s'$*, means that the system goes from state* s *to state* s' *exchanging the action* a *with the environment.*

For model checking purposes, S and A must be finite. LTSs are particular suitable for describing the dynamic of the system.

Kripke transition systems (KTSs) are a combination of KSs and LTSs, that exploit the strengths of both formalisms. KTSs are defined as follows.

[1]In literature, different definitions of model use the term *transition system* for identifying different notations. In [100], for example, *Kripke structures* are identified as transition systems, whereas in [131] transition systems identify *labeled transition systems.* We adhere to the definitions given in [135, 26], that are more specific in distinguishing the different notations.

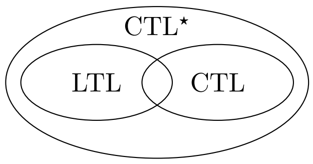


Figure 9.1: Relationship between LTL, CTL and CTL*

Definition 9.2 (Kripke transition system)**.** *A* Kripke transition system *is defined by the 5-tuple* $\langle S, S_0, A, \rightarrow, \mathcal{L}\rangle$*, being* S*,* S_0 *and* A *defined as in Def. 9.1, and* $\mathcal{L}$ *defined as in Def. 3.1.*

Model checkers notations Each model checker provides its own input syntax to encode a model. Usually these syntaxes are dialects/extensions of C, Java or VHDL. In Section 3.2 we have described the syntax of NuSMV and seen the relationship between a NuSMV model and the Kripke structure it describes. Another notation that we have used in our works is Promela, the input syntax of SPIN [99]: it is a C-like language that permits to easily describe concurrent systems.

9.1.2 Property specification

Properties that are checked in model checking are usually described in temporal logics. Temporal logics permits to describe properties not only related to the state, but that also involve the dynamic of the system over time. Temporal logics permit to describe different kind of properties [16]:

- *reachability* properties check that it is possible to reach a given state;
- *safety* properties check that *something bad never happens*;
- *liveness* properties check that *something good will eventually happen*;
- *fairness* properties check that, under some conditions, an event can repeatedly happen.

Temporal logics are divided into:

- *Linear-time logics* represent time as sequences of instants.
- *Branching-time logics* represent time as a tree, where the root is the initial instant and its children the possible evolutions of the system; it is possible to declare properties concerning all the paths or only some of them.

In Section 9.1.2.1 we describe the linear-time logic LTL, and in Section 9.1.2.2 the branching-time logic CTL. The two logics have different expressivenesses; some properties can be expressed in LTL but not in CTL and the other way around. The logic CTL* is an extension of CTL and it subsumes both CTL and LTL. The relationship between the expressivenesses of the three logics is shown in Fig. 9.1.

9.1.2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) [149] is a linear-time logic whose syntax is described by the following Backus-Naur form (BNF):

$$\varphi \ ::= \ \mathit{true} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}(\varphi) \mid \varphi \mathbf{U} \varphi$$

where p belongs to the set of atomic propositions AP.

LTL formulas can be interpreted over infinite paths of Kripke structures as described by the following definition.

Definition 9.3 (LTL interpretation over paths)**.** *Given a Kripke structure $\mathcal{M} = \langle S, S^0, T, \mathcal{L} \rangle$, an infinite path $\pi = s_1, s_2, \ldots$ in $\mathcal{M}$, and an LTL formula φ, the problem $\pi \models \varphi$ can be described using the following inductive definition*

$$\begin{array}{lll}
\pi \models true & & \\
\pi \models p & \iff & p \in \mathcal{L}(s_1) \\
\pi \models \neg\varphi & \iff & \pi \not\models \varphi \\
\pi \models \varphi_1 \wedge \varphi_2 & \iff & \pi \models \varphi_1 \wedge \pi \models \varphi_2 \\
\pi \models \mathbf{X}(\varphi) & \iff & \pi^2 \models \varphi \\
\pi \models \varphi_1 \, \mathbf{U} \, \varphi_2 & \iff & \exists i \geqslant 1 \colon (\pi^i \models \varphi_2 \wedge \forall j \in [1, i-1] \colon \pi^j \models \varphi_1)
\end{array}$$

where p is an atomic proposition taken from AP, and π^i the tail of the path starting in s_i (i.e., $\pi^i = s_i, s_{i+1}, \ldots$).

The definition says that every path satisfies *true*, and that a path satisfies an atomic proposition p iff p belongs to the labels of the first state of the path. The definitions for the negation and the conjunction are straightforward; other boolean connectives can be introduced starting from the basic ones (e.g., $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$).

The *next* operator $\mathbf{X}(\varphi)$ requires that the property φ holds starting from the *next* state, i.e., the second state of the path.

The *strong until* operator $\varphi_1 \, \mathbf{U} \, \varphi_2$ requires that φ_1 holds continuously until φ_2 holds. The operator is *strong* since it requires that φ_2 eventually holds.

Other LTL temporal operators can be defined starting from the *next* and the *strong until* operators:

- $\mathbf{F}(\varphi) = true \, \mathbf{U} \, \varphi$. The operator *finally* requires that φ eventually holds in a state of the path.

- $\mathbf{G}(\varphi) = \neg\mathbf{F}(\neg\varphi)$. The operator *globally* requires that φ holds in every state of the path.

- $\varphi_1 \, \mathbf{W} \, \varphi_2 = \varphi_1 \, \mathbf{U} \, \varphi_2 \vee \mathbf{G}(\varphi_1)$. The operator *weak until*, is a weaker version of the *strong until* operator, since it does not require that φ_2 eventually holds. If φ_2 never holds, then φ_1 must globally hold.

- $\varphi_1 \, \mathbf{R} \, \varphi_2 = \neg(\neg\varphi_1 \, \mathbf{U} \, \neg\varphi_2)$. The operator *release* requires that φ_2 must hold up to and including the state in which φ_1 becomes true. Note that φ_1 is not required to eventually hold; if φ_1 never becomes true, then φ_2 must globally hold.

In order to verify if a model $\mathcal{M}$ satisfies an LTL formula φ we must give a further definition.

Definition 9.4 (LTL model checking)**.** *Let $\mathcal{M} = \langle S, S^0, T, \mathcal{L} \rangle$ be a model, $s \in S$ a state of the model, and φ an LTL formula. The model $\mathcal{M}$ satisfies the formula φ starting from the state s if the formula is satisfied for every execution path starting from s, i.e.,*

$$\mathcal{M}, s \models \varphi \quad \iff \quad \forall \pi \in \Pi_s \colon \pi \models \varphi$$

where Π_s is the set of infinite paths starting from s.

9.1.2.2 Computation Tree Logic

Computation Tree Logic (CTL) [50] is a branching-time logic whose syntax is described by the following BNF:

$$\varphi \ ::= \ true \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{EX}(\varphi) \mid \mathbf{E}\,[\varphi \, \mathbf{U} \, \varphi] \mid \mathbf{A}\,[\varphi \, \mathbf{U} \, \varphi]$$

where p belongs to the set of atomic propositions AP. Each CTL temporal operator is composed by a pair of symbols:

1. the first symbol is **A** or **E**, meaning that the property must hold *along* **A***ll paths*, or it must **E***xist a path* in which it holds.

2. the second symbol is **X**, **U**, **F** or **G**, having the same meaning of the corresponding LTL operators (see Section 9.1.2.1).

In the following we give the semantics of the formulae built using the operators introduced in the BNF, then we derive the others operators (**AX**, **EF**, **AF**, **EG** and **AG**) starting from the basic ones.

Definition 9.5 (CTL model checking). *Given a Kripke structure $\mathcal{M} = \langle S, S^0, T, \mathcal{L} \rangle$, a state $s \in S$, and a CTL formula φ, the* CTL model checking problem $\mathcal{M}, s \models \varphi$ *can be described using the following inductive definition*

$$
\begin{array}{lll}
\mathcal{M}, s \models \mathit{true} & & \\
\mathcal{M}, s \models p & \iff & p \in \mathcal{L}(s) \\
\mathcal{M}, s \models \neg\varphi & \iff & \mathcal{M}, s \not\models \varphi \\
\mathcal{M}, s \models \varphi_1 \wedge \varphi_2 & \iff & \mathcal{M}, s \models \varphi_1 \wedge \mathcal{M}, s \models \varphi_2 \\
\mathcal{M}, s \models \mathbf{EX}(\varphi) & \iff & \exists s' \in \mathit{next}(s) \colon \mathcal{M}, s' \models \varphi \quad \text{where } \mathit{next}(s) = \{s' \in S \colon (s, s') \in T\} \\
\mathcal{M}, s \models \mathbf{E}\,[\varphi_1 \,\mathbf{U}\, \varphi_2] & \iff & \exists \pi \in \Pi_s \colon \pi \models \varphi_1 \,\mathbf{U}\, \varphi_2 \\
\mathcal{M}, s \models \mathbf{A}\,[\varphi \,\mathbf{U}\, \varphi] & \iff & \forall \pi \in \Pi_s \colon \pi \models \varphi_1 \,\mathbf{U}\, \varphi_2
\end{array}
$$

where p is an atomic proposition taken from AP, $\pi = s_1, s_2, \ldots$ is an infinite path in $\mathcal{M}$, and Π_s the set of infinite paths starting from s.

The definition says that every state satisfies *true*, and that a state satisfies an atomic proposition p iff p belongs to the labels of the state. As for LTL, the definitions for the negation and the conjunction are straightforward and other boolean connectives can be introduced starting from the basic ones.

The *existential next* operator $\mathbf{EX}(\varphi)$ requires that φ holds in at least one next state of s.

The *existentially quantified strong until* operator $\mathbf{E}\,[\varphi_1 \,\mathbf{U}\, \varphi_2]$ requires that it *exists a path* starting from s where φ_1 holds continuously until φ_2 holds (see Def. 9.3).

The *universally quantified strong until* operator $\mathbf{A}\,[\varphi_1 \,\mathbf{U}\, \varphi_2]$ requires that, *along all paths* starting from s, φ_1 holds continuously until φ_2 holds.

Other CTL temporal operators can be defined starting from the previous ones:

- $\mathbf{AX}(\varphi) = \neg\mathbf{EX}(\varphi)$. The operator requires that φ holds in *all* the *next* states of s.

- $\mathbf{EF}(\varphi) = \mathbf{E}\,[\mathit{true}\,\mathbf{U}\,\varphi]$. The operator requires that it *exists* a state in the *future* (a state belonging to a path starting from s) where φ holds.

- $\mathbf{AF}(\varphi) = \mathbf{A}\,[\mathit{true}\,\mathbf{U}\,\varphi]$. The operator requires that, in *all* the paths starting from s, it exists a state where φ holds.

- $\mathbf{EG}(\varphi) = \neg\mathbf{AF}(\varphi)$. The operator requires that it *exists* a path starting from s in which φ *globally* holds.

- $\mathbf{AG}(\varphi) = \neg\mathbf{EF}(\varphi)$. The operator requires that φ *globally* holds in *all* the paths starting from s.

9.2 Model checking ASMs

In Section 2.1 we have already underlined the advantages of using ASMs for system modeling. They permit to easily describe complex systems at different level of abstractions. As a consequence, all the concrete syntaxes developed for the ASMs (e.g., AsmetaL, CoreASM [68] and AsmL [14]) are high-level notations, general enough to describe a wide range of systems (both software and hardware), and that permit to write models that can be understood by all the stakeholders of the system under development. These notations distinguish from those used by formal verification tools (e.g., Promela, the input language of SPIN, or the input syntax of NuSMV) that are usually very low-level, sometimes difficult to understand for non-expert people, and so not usable for design and documentation purposes.

Several works [125, 11, 95] have underlined the necessity of doing formal verification directly on high-level models of the system under analysis. This would mainly have two advantages:

a) declaring a property on a high level-model of the system is definitely easier than writing the same property on a low-level model;

b) if formal verification can be done on the high-level model, the user does not have to write a low-level model only for this task, so avoiding two problems:
    - the low-level model could be more difficult to write;
    - if two different models are written using two different notations, there is the problem of proving that the two models are conformant.

Moreover, we believe that a developing environment where several tools can be used for different purposes on the base of the same specification model can be much more convenient than having different tools working on input models with their own different languages.

We here present a work that addresses the second aim of the thesis described in Section 1.2, i.e., the promotion of the integration of different FMs [34]. We report our experience in the integration of a model checker in ASMETA (see Section 2.2). Given our experience on some case studies (e.g., the Mondex protocol, see Section 9.2.2.1), having a model checker integrated with a powerful simulator provides great advantages for model analyses, especially in order to perform model and property validation. Indeed, verification of properties should be applied when a designer has enough confidence that the specification and the properties themselves capture all the informal requirements. By simulation (interactive or scenario driven as described in Chapter 4) and model review (see Chapter 7), it is possible to ensure that the specification really reflects the intended system behaviour. Otherwise there is the risk that proving properties becomes useless, for example when a property is *vacuously* true (see Section 5.2.4). Moreover, a simulator can help to reproduce counterexamples provided by a model checker, which are sometimes hermetic to understand, as suggested in [95].

The problem that arises is how to model check ASMs. Although it is relatively clear that the ASMs specification formalism enables a more convenient modeling than that provided by, for example, the language of a model checker as NuSMV, on the other hand it is undoubted that a lower-level formalism usually leads to more efficient model checking. So we claim that, rather than developing a model checker tailored on ASMs, it is more convenient to provide a mapping from the syntax of ASMs to the low-level syntax of an existing model checker.

There are several attempts to translate ASM specifications to the languages of different model checkers. For explicit state model checkers as Spin, we can cite [77] and [69]. In [77], the authors show how to obtain Promela programs from simple ASMs in order to use Spin for test generation; the approach has some limitations since it only supports 0-ary functions and does not support the choose rule[2]. The approach has been significantly improved in [69] where support has been added for arbitrary n-ary functions, for the choose rule and for distributed ASMs. The authors report their experience in using Spin for verifying properties of CoreASM specifications on the FLASH Cache Coherence Protocol.

In [168], the author discusses the use of the model checker SMV (Symbolic Model Verifier) in combination with ASMs. A scheme is introduced for transforming ASM models into the language of SMV from ASM workbench specifications. The approach was later improved in [40] and applied

[2]Actually, in the work described in Chapter 12, the tool presented in [77] has been extended to support the choose rule.

to a complex case study in [169]. In this approach ASMs are iteratively flattened in a series of conditional rules that can be easily mapped in SMV. In [168] only 0-ary functions are supported, whereas in [40] also generic n-ary functions are supported. The approach is similar to the scheme we present here in Section 9.2.1 since the target notation is similar (SMV/NuSMV). However, our approach does not need to flatten the ASM for the translation, and natively supports the choose rule; in their approach, instead, the choose rules must be substituted by appropriate monitored functions.

Other approaches to model check ASMs include works which perform a quasi-native model checking without the need of a translation to a different notation. For example, [111] presents a model checking algorithm for AsmL specifications. The advantages is that the input language is very rich and expressive, but the price is that the model checking is very inefficient and unable to deal with complex specifications, and it is not able to perform all the optimizations available for a well established technique as that of Spin or NuSMV. A mixed approach is taken by [23], which presents a way for model checking ASMs without the need of translating ASM specifications into the modeling language of an existing model checker. Indeed, they combine the model checker [mc]square with the CoreASM simulator which is used to build the state space.

A different approach is presented in [160], where Answer Set Programming is used for doing bounded model checking of ASMs.

We here present *AsmetaSMV*, a tool that enriches the ASMETA framework with the capabilities of the model checker NuSMV [137]. AsmetaSMV, besides the traditional verification purposes, is also used by another tool of the ASMETA framework (AsmetaMA) to do model review of ASM specifications (see Chapter 7).

In Section 9.2.1 we describe the general architecture of AsmetaSMV and the process of automatically mapping ASM models into NuSMV models. In Section 9.2.2, we report the results of using AsmetaSMV to verify temporal properties of various case studies of different characteristics and complexity.

9.2.1 AsmetaSMV

AsmetaSMV has been developed as *based* tool of the ASMETA toolset, since it exploits some derivatives of the ASMETA environment. In particular, AsmetaSMV does not define its own input language, neither introduces a parser for a textual syntax. It reuses the parser defined for AsmetaL and reads the models as Java objects as defined by the ASMETA Java API. The aim of AsmetaSMV is that of enriching the ASMETA toolset with the capabilities of the model checker NuSMV. No knowledge of the NuSMV syntax is required to the user in order to use AsmetaSMV. To perform model checking over ASM models written in AsmetaL, a user must know, besides the AsmetaL language, only the syntax of the temporal operators.

AsmetaSMV supports a wide range of ASMs, both single and multi-agents (synchronous and asynchronous); limitations are due to the model checker restriction over finite domains and data types.

Fig. 9.2 shows the general architecture of the tool. We assume that the user provides a model in AsmetaL, but any other concrete syntax (like Asmeta XMI) could be used instead. The tool parses the model and then checks if it is adequate to be mapped into NuSMV. If this test fails, an exception is risen; otherwise, signature and transitions rules are translated as described in Section 9.2.1.1 and 9.2.1.2. The user can define temporal properties directly into the AsmetaL code as described in Section 9.2.1.3. The invocation of the model checker can be separately done on the translated model or directly from AsmetaSMV (see Section 9.2.1.4).

9.2.1.1 Mapping of the signature

Domains The AsmetaL domains that can be mapped in NuSMV are only those that have a corresponding type in NuSMV and that are finite. The supported domains are: Boolean, Enum domains, Concrete domains whose type domains are Integer or Natural, and abstract domains.

Boolean and Enum domains are straightforwardly mapped into boolean and symbolic enu-

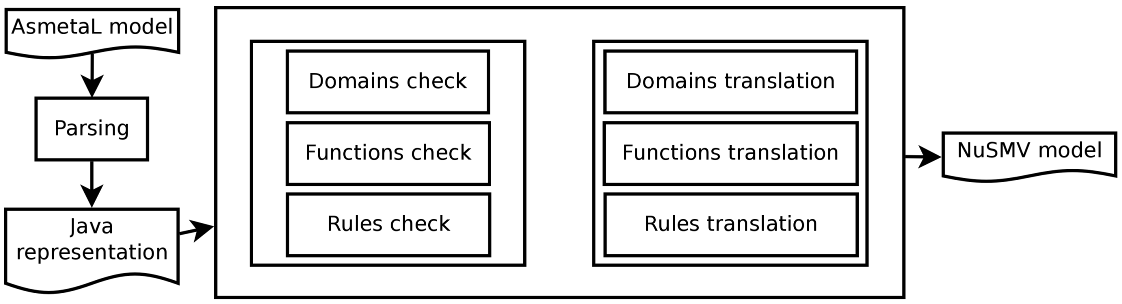


Figure 9.2: Architecture of AsmetaSMV

```
asm example
import StandardLibrary

signature:
    domain SubDom subsetof Integer
    enum domain EnDom = {AA | BB}
    controlled foo0: Boolean
    controlled foo1: Boolean −> EnDom
    controlled foo2: Prod(SubDom , EnDom) −> SubDom

definitions:
    domain SubDom = {1..2}
```

```
MODULE main
    VAR
        foo0: boolean;
        foo1_FALSE: {AA, BB, ENDOM_UNDEF};
        foo1_TRUE: {AA, BB, ENDOM_UNDEF};
        foo2_1_AA: {1, 2, −2147483647};
        foo2_1_BB: {1, 2, −2147483647};
        foo2_2_AA: {1, 2, −2147483647};
        foo2_2_BB: {1, 2, −2147483647};
```

Figure 9.3: Example of domains and functions mapping – AsmetaL and NuSMV models

merative types of NuSMV. Concrete domains of Integer and Natural, instead, become integer enumeratives in NuSMV, on the base of the concrete domain definitions. Abstract domains are mapped into enumerative domains. Abstract domains must be static, i.e., they can not be extended using the *extend* rule, since we need that the state space of the machine must be fixed and finite.

All the domains previously introduced are supported both as function domains and as function codomains. There is another domain that, instead, is only supported when it is used as function domain, the *product* domain. Product domains are used as function domains to define n-ary functions with $n \geqslant 2$.

For almost all the domains, when these are mapped into functions codomains, we add a special value that represents the *undef* value of ASMs. For enumeratives domains we add a new enumerative value, using the format *DOMNAME*_UNDEF where *DOMNAME* is the uppercase version of the name of the domain. For integers domains we add the minimum value of NuSMV integers[3]. Only for the Boolean domain we can not provide the undef value in NuSMV, since we decided to map it directly to the boolean type of NuSMV; so, we require that boolean functions are always defined. A solution could be to translate the Boolean domain in an enumerative type with three values, but this approach would complicate too much the translation of boolean expressions.

In Fig. 9.3 we can see how the domains *SubDom* and *EnDom* have been mapped. We see that, when they are used as function codomains, also the mapping of the *undef* value is added. In Section 9.3 we will explain what is the effect of the domain mapping when the domain is used as a function domain.

[3]In case the minimum value of NuSMV integers is already a value of the domain, we look for another suitable value.

```
asm ex
import StandardLibrary

signature:
    enum domain EnDom = {AA | BB}
    dynamic monitored mon: Boolean
    dynamic controlled contr: EnDom

definitions:
    main rule r_Main =
        if mon then
            contr := AA
        else
            contr := BB
        endif

default init s0:
    function contr = AA
```

```
MODULE main
    VAR
        mon: boolean;
        contr: {AA, BB, ENDOM_UNDEF};
    ASSIGN
        init(contr) := AA;
        next(contr) :=
            case
                mon: AA;
                !mon: BB;
                TRUE: contr;
            esac;
```

Figure 9.4: Example of controlled and monitored functions mapping – AsmetaL and NuSMV models

Functions For each AsmetaL dynamic *nullary* function (i.e., variable) a NuSMV variable is created. ASM *n-ary* functions (with $n > 0$), instead, must be decomposed into function locations; each location is mapped into a NuSMV variable. So, the cardinality of the domain of a function determines the number of the corresponding variables in NuSMV. Therefore, given an 1-ary function with domain D_1, or an n-ary function (with $n \geqslant 2$) with domain $Prod(D_1, \ldots, D_n)$, in NuSMV we introduce $\prod_{i=1}^{n} |D_i|$ variables with names

$$\mathit{func\_elDom}_1\_\ldots\_\mathit{elDom}_n$$

where $elDom_1 \in D_1, \ldots, elDom_n \in D_n$, and *func* is the function name.

As seen in Section 9.2.1.1, the mapping of the codomain of a function determines the type of the variable.

In Fig. 9.3 the 0-ary function *foo0* has been mapped to a single variable, the unary function *foo1* has been mapped to two variables, since the function has two locations, and *foo2* has generated four variables, since it is composed of four locations.

Controlled functions Controlled functions are the only functions whose value can be updated in a transition rule. The initialisation and the updates of each controlled location are mapped in the ASSIGN section of NuSMV through the *init* and *next* instructions. If a location is not initialised, it is initialised in NuSMV to the corresponding value of *undef* for its codomain.

The computation of the conditions under which a location must be updated will be explained in Section 9.2.1.2. Here, we can simply say that all the updates of a location in the AsmetaL model are collected together and executed through a case expression in NuSMV. A default condition is always added for keeping the value of the location unchanged when no update of the location fires.

In Fig. 9.4, the function *contr* is an example of controlled function that is initialised in the initial state, and updated by two different update rules, based on the value of the monitored function *mon*. Note that in this case the default condition is not necessary, since the first two conditions are complete, i.e., in each state one of the two conditions is satisfied. In this particular simple example, NuSMV permits to omit the default condition, since it understands that the first two conditions are complete; but usually it forces the user to add the default condition, even if the conditions are complete (see Section 3.2.1 for more details).

```
asm statDer
import StandardLibrary

signature:
    domain ConcrDom subsetof Integer
    dynamic monitored mon1: Boolean
    dynamic monitored mon2: Boolean
    static stat: ConcrDom
    derived der: ConcrDom

definitions:
    domain ConcrDom = {1..4}

    function stat = 2

    function der =
        if mon1 then
            1
        else if mon2 then
            3
        endif endif
```

```
MODULE main
    VAR
        mon1: boolean;
        mon2: boolean;
    DEFINE
        stat := 2;
        der :=
            case
                mon1: 1;
                !mon1 & mon2: 3;
                TRUE: −2147483647;
            esac;
```

Figure 9.5: Example of static and derived functions mapping – AsmetaL and NuSMV models

Monitored functions Monitored functions are functions whose value is set by the environment. In NuSMV, monitored variables are declared but they are neither initialised nor updated. When NuSMV meets a monitored variable it creates a state for each value of the variable.

In Fig. 9.4 the corresponding variable of the monitored function *mon* is only declared, but neither initialised nor updated.

Static and derived functions Static and derived functions can not be updated either by an update rule or by the environment; their value (for static functions) or their computation mechanism (for derived functions) is defined in the section *definitions* and never changes during the execution of the machine. AsmetaSMV does not distinguish between static and derived functions: their mapping is the same. In the target NuSMV model, static and derived functions are expressed through the DEFINE statement. Let's remember that a DEFINE declaration does not introduce new variables, but acts as a macro that introduces aliases for (complex) expressions.

The mapping of a static/derived location requires to associate, at each term t_i to which the location can be defined, the corresponding condition c_i under which the definition takes place. In NuSMV, all the couples (c_i, t_i) are reported through a case expression in the corresponding definition. Since in ASMs a not defined static/derived location takes value *undef*, we add to the case expression a default condition that sets the definition to the undef value of the function codomain (see Section 9.2.1.1).

In Fig. 9.5 the mapping of the static function *stat* is straightforward, whereas the mapping of the derived function *der* is more complex. The definition of the function *der* has been visited and the conditions under which it takes value 1 or 3 have been computed and mapped in the case expression of the definition in NuSMV. Since the ASM function is not total, in the case expression the default condition sets the definition to the undef value of integers.

9.2.1.2 Mapping of transition rules

ASMs and NuSMV differ in the way they compute the next state of a transition, and such difference is reflected in their syntaxes as well.

In ASMs, at each state, every enabled rule is evaluated and the update set is built by collecting all the locations and next values to which locations must be updated. The same location can be updated to different values in several points of the specification under different conditions.

In NuSMV, at each step, for every variable, the next value is computed by considering *all* its possible guarded assignments. As we have seen in Section 3.2, the next value of a variable is defined once, listing all the guarded assignments. For example, the next value of variable *var* could be defined using a case expression in the following way

```
next(var) :=
    case
        cond1: val1;
        cond2: val2;
         ...
    esac;
```

where all the possible next values for the variable are listed.

So, for our purposes, the main difference between ASMs and NuSMV is that in ASMs the updates of a location can be distributed all over the model, whereas in NuSMV the definition of the next state of a variable is grouped in a single expression. In order to map transition rules from AsmetaL to NuSMV, our translation algorithm visits the ASM specification looking for all the updates and arranging them as required by NuSMV. It starts from the main rule and, by executing a depth visit of all the rules it encounters, it builds a *conditional update map*, which maps every location to its update value together with its guard. A global stack *Conds* (initialised to *true*) is used to store the conditions of all the outer rules visited. For each transition rule, a suitable visit procedure has been defined. In the following we briefly overview the mapping of the update, condition and choose rules, and finally list all the other rules that are supported by our translation algorithm.

Update rule The update rule syntax is

$$l := t$$

where l is a location and t a term.

The translation algorithm builds c as the conjunction of all the conditions on the stack *Conds* and adds to the *conditional update map* the triple (l, c, t), specifying that location l is updated to t under condition c.

Conditional Rule The conditional rule syntax is

```
if cond then
    $R_{then}$
[else
    $R_{else}$]
endif
```

where *cond* is a boolean expression and R_{then} and R_{else} are transition rules.

The translation algorithm works as follows:

1. *cond* is put on stack *Conds* and rule R_{then} is visited; in such a way all the updates contained in R_{then} are executed only if *cond* is true;
2. *cond* is removed from stack *Conds*.
3. If the **else** branch is not null:
   (a) condition $\neg cond$ is put on stack *Conds* and rule R_{else} is visited; in such a way all the updates contained in R_{else} are executed only if *cond* is false;
   (b) $\neg cond$ is removed from stack *Conds*.

For example, the conditional update map that results from the visits of the AsmetaL model shown in Fig. 9.6 is the following

```
asm condRule
import StandardLibrary
signature:
    enum domain EnDom = {AA| BB| CC}
    dynamic monitored mon: Boolean
    dynamic monitored mon2: Boolean
    dynamic controlled contr: EnDom
    dynamic controlled contr1: EnDom

definitions:
    main rule r_Main =
        par
            contr1 := AA
            if(mon) then
                if(mon2) then
                    contr := BB
                else
                    contr := AA
                endif
            endif
        endpar
```

```
MODULE main
    VAR
        contr: {AA, BB, CC, ENDOM_UNDEF};
        contr1: {AA, BB, CC, ENDOM_UNDEF};
        mon: boolean;
        mon2: boolean;
    ASSIGN
        init(contr) := ENDOM_UNDEF;
        init(contr1) := ENDOM_UNDEF;
        next(contr) :=
            case
                mon & mon2: BB;
                mon & !mon2: AA;
                TRUE: contr;
            esac;
        next(contr1) := AA;
```

Figure 9.6: Example of conditional rule mapping – AsmetaL and NuSMV models

Location	**Condition**	**Value**
contr	mon $\wedge$ mon2	BB
	mon $\wedge$ $\neg$ mon2	AA
contr1	*true*	AA

Choose rule The choose rule syntax is

$$\begin{aligned}&\textbf{choose } v_1 \textbf{ in } D_1, \dots, v_n \textbf{ in } D_n \textbf{ with } G(v_1, \dots, v_n) \textbf{ do}\\ &\qquad R(v_1, \dots, v_n)\\ &\quad [\textbf{ifnone } R_{ifnone}]\end{aligned}$$

where $v_1, \dots, v_n$ are logical variables and $D_1, \dots, D_n$ their domains. $G(v_1, \dots, v_n)$ is a boolean condition, and $R(v_1, \dots, v_n)$ a transition rule. Optional branch **ifnone** contains the rule R_{ifnone} that must be executed if there are no values for variables $v_1, \dots, v_n$ that satisfy $G(v_1, \dots, v_n)$.

In the mapping process, each choose rule is identified by an identifier *chId*. In NuSMV, for each variable v_i, a variable $lv\_v_i\_chId$ is created; the type of such variable is obtained with the mapping of domain D_i. The translation algorithm, for each tuple of values $(d_1, \dots, d_n) \in D_1 \times, \dots, \times D_n$, executes the following operations:

1. it adds to the stack *Conds* the condition $cond_j$, built as follows

$$cond_j = \bigwedge_{i=1}^{n} lv\_v_i\_chId = d_i \wedge G(v_1 \leftarrow d_1, \dots, v_n \leftarrow d_n)$$

2. rule $R(v_1 \leftarrow d_1, \dots, v_n \leftarrow d_n)$ is visited;

3. condition $cond_j$ is removed from stack *Conds*.

The number of iterations of this process is $nB = \prod_{i=1}^{n} |D_i|$.

Finally, if the branch **ifnone** is not null, the algorithm executes the following operations:

```
asm chooseRule
import StandardLibrary

signature:
    domain ConcrDom subsetof Integer
    dynamic controlled foo: ConcrDom

definitions:
    domain ConcrDom = {1..4}

    main rule r_Main =
        choose $x in ConcrDom with $x < 3 do
            foo := $x + 2
          ifnone
              foo := 1

default init s0:
    function foo = 1
```

```
MODULE main
   VAR
      foo: {1, 2, 3, 4, −2147483647};
      lv_$x_0: 1..4;
   ASSIGN
      init(foo) := 1
      next(foo) :=
         case
            lv_$x_0 = 1 & 1 < 3: 1 + 2;
            lv_$x_0 = 2 & 2 < 3: 2 + 2;
            lv_$x_0 = 3 & 3 < 3: 3 + 2;
            lv_$x_0 = 4 & 4 < 3: 4 + 2;
            !(1 < 3) & !(2 < 3) & !(3 < 3) & !(4 < 3): 1;
            TRUE: foo;
         esac;
   INVAR (lv_$x_0 = 1 & 1 < 3) | (lv_$x_0 = 2 & 2 < 3) |
           (lv_$x_0 = 3 & 3 < 3) | (lv_$x_0 = 4 & 4 < 3) |
           (!(1 < 3) & !(2 < 3) & !(3 < 3) & !(4 < 3));
```

Figure 9.7: Example of choose rule mapping – AsmetaL and NuSMV models

1. it adds to the stack *Conds* the condition *ifNoneCond*, built as follows

$$\mathit{ifNoneCond} = \bigwedge_{\substack{(d_1,\ldots,d_n)\in\\ D_1\times\ldots\times D_n}} \neg G(v_1 \leftarrow d_1, \ldots, v_n \leftarrow d_n)$$

2. rule R_{ifnone} is visited;

3. condition *ifNoneCond* is removed from stack *Conds*.

The procedure we have described permits to add the right conditions to the updates contained in the scope of the choose rule.

Now, we must render the semantics of the choose rule in NuSMV. We must assure that, in each state, a condition $cond_j$ (with $j = 1, \ldots, nB$) is satisfied or, if this is not possible, that condition *ifNoneCond* is satisfied. To this purpose, we define the following invariant in the INVAR section:

$$\bigvee_{j=1}^{nB} cond_j \vee \mathit{ifNoneCond}$$

The invariant, if possible, bounds the logical variables of the choose rule to those values that satisfy the guard of the rule. Note that we must also add to the invariant the condition *ifNoneCond* so that the invariant is true also when no $cond_j$ is true.

In Fig. 9.7 there is an example of ASM with choose rule. We can see that the choose rule has been decomposed into:

- a variable $lv\_\$x\_0$ that corresponds to the logic variable $\$x$ of the choose rule;

- four branches in the case expression that computes the next value of variable *foo*. The conditions depend on the value of $lv\_\$x\_0$ and on the guard of the choose rule. An additional branch of the case expression corresponds to the **ifnone** rule of the choose rule;

- an INVAR declaration that bounds $lv\_\$x\_0$ in the range $[1, 2]$.

Actually, the model produced by AsmetaSMV is simpler of that shown in Fig. 9.7. In fact the tool, where possible, evaluates boolean and integer expressions[4]. The simplified model is

```
MODULE main
  VAR
    foo: {1, 2, 3, 4, −2147483647};
    lv_$x_0: 1..4;
  ASSIGN
    init(foo) := 1;
    next(foo) :=
      case
        lv_$x_0 = 1: 3;
        lv_$x_0 = 2: 4;
        TRUE: foo;
      esac;
  INVAR lv_$x_0 = 1 | lv_$x_0 = 2;
```

Other rules In addition to the *update*, *conditional* and *choose* rules, the other transition rules that are supported by AsmetaSMV are: *skip*, *macrocall*, *block*, *case*, *let* and *forall* rules. Details can be found in [5].

We do not support the *extend* rule since the state space of the machine must be fixed and finite. A solution could be to require that an extend rule is called a fixed number of times: in this case we could add in the NuSMV model all the elements of the *reserve* that can be added, and use a characteristic function to translate the extend rule.

We do not support *turbo* rules since their translation would be too complicated. We only support simple forms of the *seq* rule.

9.2.1.3 Temporal property specification

AsmetaSMV allows the user to declare CTL/LTL properties directly in the AsmetaL model, before the main rule. The syntax of a CTL/LTL property in AsmetaL is:

CTLSPEC/LTLSPEC p

where *p* is a boolean expression possibly containing boolean functions that represents the temporal operators of CTL/LTL. In order to write temporal formulae in AsmetaL, we have developed the libraries *CTLlibrary.asm* and *LTLlibrary.asm* where, for each CTL/LTL operator supported by NuSMV, an equivalent boolean function is declared.

In formal verification there are commonly used *patterns* that are formed by the combination of the basic operators: in CTL, for example, the *global precedence* pattern requires that s always precedes p, i.e., $\mathbf{A}[\neg p \ \mathbf{W} \ s]$. In [61] a lot of patterns for CTL and LTL have been proposed. We have added some of them to our libraries: this let the user write complex specifications in a compact way.

Table 9.1 reports some of the CTL functions contained in the library *CTLlibrary.asm.* Note that the *weak until* operators have been added in the pattern section, since they are not natively supported by NuSMV.

Fig. 9.8 contains a fragment of the AsmetaL model of the dining philosophers problem. It contains a reachability property that checks that it is possible to reach a state in which the first and the third philosophers are eating; also the negation of the property has been added in order to obtain a counterexample. Moreover, two liveness properties check that each philosopher can always eat and think. Thanks to the AsmetaL *forall term*, the two liveness properties can be written very concisely, specifying the same property for all the philosophers; in the NuSMV model, they are unfolded as conjunctions.

9.2.1.4 Temporal property verification

AsmetaSMV allows model checking an AsmetaL specification by translating it to the NuSMV language and directly run the NuSMV tool on this translation to verify the properties.

[4]It is possible to avoid the simplification using the execution option *-ns*.

NuSMV CTL operator	**AsmetaL CTL function**
EG p	static eg: Boolean → Boolean
EX p	static ex: Boolean → Boolean
EF p	static ef: Boolean → Boolean
AG p	static ag: Boolean → Boolean
AX p	static ax: Boolean → Boolean
AF p	static af: Boolean → Boolean
E[p **U** q]	static e: Prod(Boolean, Boolean) → Boolean
A[p **U** q]	static a: Prod(Boolean, Boolean) → Boolean

CTL pattern	**AsmetaL CTL function**
E[p **W** q]	static ew: Prod(Boolean, Boolean) → Boolean
A[p **W** q]	static aw: Prod(Boolean, Boolean) → Boolean
A[¬p **W** q]	static ap: Prod(Boolean, Boolean) → Boolean
A[(¬p ∨ **AG**(¬r)) **W** (q ∨ r)]	static pb: Prod(Boolean, Boolean, Boolean) → Boolean
...	...

Table 9.1: CTL operators and CTL patterns available in CTLlibrary.asm

The output produced by NuSMV is pretty-printed, replacing the NuSMV variables with the corresponding AsmetaL locations: it is our desire, in fact, to hide as much as possible the NuSMV syntax to the user.

Fig. 9.9 reports the output produced by model checking the AsmetaL model shown in Fig.9.8. All the three properties are verified. The negation of the reachability property shows a path to reach a state in which the first and the third philosophers are eating. Note that the pretty printer has substituted variables with locations (e.g., *eating_PHIL1* with *eating(PHIL1)*).

Thanks to the integration of AsmetaSMV in the ASMETA framework, a counterexample can be mapped to an Avalla scenario (see Section 4.2), so that the trace that leads to a failure can be reproduced through simulation. This feature has also been encouraged in [95], where it is noticed that a good formal method framework should provide a better feedback when formal analysis exposes a failure, so that for the user it is easier to discover the fault and fix it.

9.2.2 Experiments

AsmetaSMV has been tested on five case studies; the complete description of our tests can be found in [5], and all the models are available in the ASMETA repository [13].

The first two case studies we have analysed are two problems described in [31]:

1. A system made of two traffic lights placed at the beginning and at the end of an alternated one-way street; both traffic lights are controlled by a computer.

2. An irrigation system composed of a small sluice, with a rising and a falling gate, and a computer that controls the sluice gate.

For both problems we have written *ground* and *refined* models; in each model we have declared safety and liveness properties to test the correctness of the model.

We have also analysed the taxi booking problem: in a city some clients can request one or more taxis to a central that must satisfy all the requests. The taxis must bring the clients where they want to go. For this problem we had previously developed a NuSMV model (let us call it *originalNuSMV*); now we have developed an AsmetaL model containing the same properties that we wrote in the *originalNuSMV*. We have been able to compare the *originalNuSMV* code with the code obtained from the translation of the AsmetaL model (let us call it *mappedNuSMV*).

```
asm philosophers
import StandardLibrary
import CTLlibrary

signature:
    ...
definitions:
    ...

    //reachability
    CTLSPEC ef(eating(phil1) and eating(phil3))
    CTLSPEC not(ef(eating(phil1) and eating(phil3)))

    //liveness
    CTLSPEC (forall $p in Philosophers with
                              ag(ef(eating($p))))
    CTLSPEC (forall $p in Philosophers with
                              ag(ef(not(eating($p)))))

    main rule r_choose_philo =
        choose $p in Philosophers with true do
            program($p)

default init s0: ...
```

```
MODULE main
    VAR
    ...
    DEFINE
    ...
    ASSIGN
    ...

--CTL properties
CTLSPEC EF(eating_PHIL3 & eating_PHIL1)
CTLSPEC !EF(eating_PHIL3 & eating_PHIL1)
CTLSPEC AG(EF(eating_PHIL1)) &
             AG(EF(eating_PHIL2)) &
             AG(EF(eating_PHIL3)) &
             AG(EF(eating_PHIL4)) &
             AG(EF(eating_PHIL5))
CTLSPEC AG(EF(!eating_PHIL1)) &
             AG(EF(!eating_PHIL2)) &
             AG(EF(!eating_PHIL3)) &
             AG(EF(!eating_PHIL4)) &
             AG(EF(!eating_PHIL5))
```

Figure 9.8: Example of temporal properties mapping – AsmetaL and NuSMV models

We have seen that, for the same problem, it is easier to write an AsmetaL code rather than a NuSMV one: the ASMs in fact, thanks to a wide set of transition rules, are much more expressive than NuSMV. The verification of the properties in *originalNuSMV* and in *mappedNuSMV* gave the same results. Obviously this cannot be considered as a demonstration of the correctness of the mapping, but shows that, for a problem, there are different equivalent models. Generally, the code obtained from a mapping is more computational onerous than a code written directly in NuSMV; the mapping, in fact, introduces some elements that can be avoided in the direct encoding.

We have applied our tool to the FLASH cache coherence protocol, which integrates support for cache coherent shared memory for a large number of interconnected processing nodes. Starting from the specifications published by Winter [169] and by Farahbod at alt. [69], we have written the AsmetaL specification for the protocol together with its safety properties. By means of the ASMETA simulator and the validator we were able to correct some defects in our specifications even before trying to prove the properties. A problem of vacuity detection has also been arisen.

Finally, we have analysed the Mondex protocol [133], as described in the next section.

9.2.2.1 Mondex protocol

The Mondex protocol [133] implements electronic cash transfer between two purses (here *cards*); the transfer of money is implemented through the sending of messages over a lossy medium, that can be a device with two slots or an Internet connection. We have analysed the first refinement described in Section 3 ("From Atomic Transfers to Messages") of [154]. The protocol that sends money from card $card_1$ to card $card_2$ works in four steps:

1. $card_2$ sends a request of money to $card_1$ with the rule STARTTO;
2. $card_1$ receives a request and sends money to $card_2$ with the rule REQ;
3. $card_2$ receives the money and sends an acknowledgement to $card_1$ with the rule VAL;
4. $card_1$ receives the acknowledgement with the rule ACK.

```
-- specification EF (eating(PHIL3) & eating(PHIL1))  is true
-- specification !(EF (eating(PHIL3) & eating(PHIL1)))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  eating(PHIL1) = FALSE
  ...
  eating(PHIL5) = FALSE
  ...
  hungry(PHIL5) = FALSE
  owner(FORK1) = undef
  ...
  owner(FORK5) = undef
  lv_$p_0 = PHIL1
-> State: 1.2 <-
  hungry(PHIL1) = TRUE
-> State: 1.3 <-
  eating(PHIL1) = TRUE
  hungry(PHIL1) = FALSE
  hungry(PHIL3) = TRUE
  owner(FORK1) = PHIL1
  owner(FORK2) = PHIL1
  lv_$p_0 = PHIL3
-> State: 1.4 <-
  eating(PHIL3) = TRUE
  hungry(PHIL3) = FALSE
  owner(FORK3) = PHIL3
  owner(FORK4) = PHIL3
  lv_$p_0 = PHIL1
-- specification AG (EF eating(PHIL1)) & ... & AG (EF eating(PHIL5))  is true
-- specification AG (EF !eating(PHIL1)) & ... & AG (EF !eating(PHIL1))  is true
```

Figure 9.9: NuSMV model execution embedded in AsmetaSMV

There are two additional rules: rule LOSEMSG models the loss of a message, and rule ABORT the abortion of a protocol run by one of the cards involved.

We have written a simplified version of the protocol in AsmetaL (Section 9.2.2.1). Since the model must be translated into NuSMV, we have only used elements that are supported by AsmetaSMV and we have slightly changed the signature.

The model presented in [154] contains an error; in Section 9.2.2.1 we describe how we found it with AsmetaSMV and we propose a solution to fix it.

Mondex model We consider a simplified version of the problem presented in [154]:

- there are only two cards (*CARD1* and *CARD2* belonging to the domain *Name*);

- it is not possible that a message is lost and that a card aborts a transaction, that is rules LOSEMSG and ABORT of [154] are not considered.

A fragment of the model is shown in Code 9.1.

```
    macro rule r_startTo($initiator in Name) =
        if( isFree( $initiator )) then
            choose $na in Name, $value in MoneyDom, $tid in TidDom with not(tids($tid)) and
                                                        authentic($na) and $na != $initiator do
                par
                    inbox($na, REQ, $initiator, $value, $tid) := true //request money
                    outboxMessage($initiator) := REQ          outboxName($initiator) := $na
                    outboxMoney($initiator) := $value         outboxTid($initiator) := $tid
                    outboxIsNone($initiator) := false
                    tids($tid) := true //tid used
                endpar
        endif

    macro rule r_req($receiver in Name) =
        choose $na in Name, $value in MoneyDom, $tid in TidDom with authentic($na) and $na!=$receiver
                                            and inbox($receiver, REQ, $na, $value, $tid)
                                            and $value <= balance($receiver) and isFree($receiver) do
            par
                inbox($receiver, REQ, $na, $value, $tid) := false //message read
                balance($receiver) := balance($receiver) − $value //subtraction of the requested money
                inbox($na, VAL, $receiver, $value, $tid) := true //reply with a VAL message
                outboxMessage($receiver) := VAL          outboxName($receiver) := $na
                outboxMoney($receiver) := $value         outboxTid($receiver) := $tid
                outboxIsNone($receiver) := false
            endpar

    macro rule r_val($receiver in Name) = ...
    macro rule r_ack($receiver in Name) = ...

    CTLSPEC ag(inbox(CARD1, REQ, CARD2, 0n, 1n) implies ef(inbox(CARD2, VAL, CARD1, 0n, 1n)))

    main rule r_irule =
        choose $card in Name, $rule in RuleId with authentic($receiver) do
            switch($rule)
                case STARTTORULE: r_startTo[$card]
                case REQRULE: r_req[$card]
                case VALRULE: r_val[$card]
                case ACKRULE: r_ack[$card]
            endswitch

default init s0:
    function balance($n in Name) = at({CARD1−>5n, CARD2−>5n}, $n)
    function inbox($n in Name, $t in MsgType, $na in Name, $value in MoneyDom, $tid in TidDom) = false
    function tids($tid in TidDom) = false
    function outboxIsNone($n in Name) = true
```

Code 9.1: Mondex protocol with error: AsmetaL model

outbox of CARD2	**inbox of CARD1**
outboxMessage(CARD2) = REQ outboxName(CARD2) = CARD1 outboxMoney(CARD2) = 0n outboxTid(CARD2) = 1n outboxIsNone(CARD2) = false	inbox(CARD1, REQ, CARD2, 0n, 1n) = true

Figure 9.10: Correspondence between the outbox sender and the inbox of the receiver

Signature We here briefly describe the signature of the model[5]. The controlled function *balance($n in Name)* represents the balance of card *$n*; the static boolean function *authentic($n in Name)* says if card $n is authentic. Each card has an inbox that contains messages that have to be processed. The controlled boolean function *inbox* models the inboxes of the cards; the arguments are:

- *$n*: the owner of the inbox;
- *$m*: the type of the message (REQ, VAL or ACK);
- *$na*: the sender of the message;
- *$value*: the amount of money involved in the transaction; to keep down the model checker execution time, its domain is limited to {0, 5, 10};
- *$t*: the identifier of the transaction.

The location *inbox($n, $m, $na, $value, $t)* is true if the message *($m, $na, $value, $t)* is in the inbox of *$n*.

Each card has an outbox that contains the last sent message. The outbox is modeled through five controlled functions that contain the elements of the message:

- *outboxMessage: Name → MsgType*: type of the message;
- *outboxName: Name → Name*: the addressee of the message;
- *outboxMoney: Name → MoneyDom*: the amount of money involved in the transaction;
- *outboxTid: Name → TidDom*: the identifier of the transaction;
- *outboxIsNone: Name → Boolean*: it signals if the outbox contains a message.

The element of the domain *Name* used as argument of the five functions is the owner of the outbox.

Fig. 9.10 reports an example to visualize the correspondence between the outbox of the card that has sent the message and the inbox of the card that has received it. In this example CARD2 has sent a REQ message of 0 money to CARD1; the identifier of the transaction is 1.

The derived function *isFree($n Name)* says if the outbox of *$n* is not involved in a transaction (the outbox contains no message or contains an ACK message). Finally, the boolean function

[5]The complete signature of the model is as follows.
enum domain Name = {CARD1 | CARD2}
enum domain MsgType = {REQ | VAL | ACK}
enum domain RuleId = {STARTTORULE | REQRULE | VALRULE | ACKRULE}
domain TidDom subsetof Natural
domain MoneyDom subsetof Natural
controlled balance: Name → MoneyDom
controlled tids: TidDom → Boolean
controlled inbox: Prod(Name, MsgType, Name, MoneyDom, TidDom) → Boolean
controlled outboxMessage: Name → MsgType
controlled outboxName: Name → Name
controlled outboxMoney: Name → MoneyDom
controlled outboxTid: Name → TidDom
controlled outboxIsNone: Name → Boolean
derived isFree: Name → Boolean
static authentic: Name → Boolean

tid($t in TidDomain) says if an identifier of transaction (tid) has already been used (since we must do model checking, we have a fixed number of tids, i.e., we can not create fresh ones).

Transition rules Let's describe the transition rules, by instantiating in our model the transfer of money from *card₁* to *card₂* described in Section 9.2.2.1:

1. *r_startTo($initiator in Name)*: *card₂* (here the *initiator*) requests v money from *card₁* (it sends a REQ message to the *inbox* of *card₁*) and memorizes the message in its outbox;
2. *r_req($receiver in Name)*: *card₁* (here the *receiver*) receives the request, removes the message from its inbox, removes v money from its balance, sends a VAL message to the *inbox* of *card₂* and puts the message also in its outbox;
3. *r_val($receiver in Name)*: *card₂* (here the *receiver*) receives the VAL message, removes the message from its inbox, adds v money to its balance, sends an ACK message to the inbox of *card₁* and puts the message also in its outbox;
4. *r_ack($receiver in Name)*: *card₁* (here the *receiver*) receives the ACK message, removes the message from its inbox and clears its outbox.

In the main rule, an authentic card and a rule are nondeterministically chosen; the chosen card executes the chosen rule. If the chosen rule is *r_startTo*, the card acts as the initiator of a communication, whereas in the other three rules the card always receives a message and, if required (i.e., in *r_req* and *r_val*), replies with a suitable message.

Formal verification We now describe how we found an error in the model presented in [154].

Let's focus our attention on the first two rules, *r_startTo* and *r_req*; we want to show that the system, through a particular execution of this two rules, can enter in a deadlock state.

The execution of rule *r_startTo($initiator in Name)* is the following:

1. the rule can be executed only if card *$initiator* is not involved in a previous transaction (i.e., *isFree($initiator)*);
2. a message *($na, REQ, $value, $tid)* is built such that card *$na* is authentic and different from card *$initiator*, and the tid *$tid* has not yet been used; if it is possible to build such a message, the following actions are executed:
   (a) the card *$initiator* puts the message in the inbox of card *$na* and in its outbox;
   (b) the tid *$tid* is removed from the available tids.

The execution of rule *r_req($receiver in Name)* is the following:

1. the rule can be executed only if card *$receiver* is not involved in a previous transaction (i.e., *isFree($receiver)*);
2. a message *($na, REQ, $value, $tid)* contained in the inbox of *$receiver* is chosen such that card *$na* is authentic and different from card *$receiver*, and the amount of money *$value* is less or equal to the balance of the receiver; if it is possible to find such a message, the following actions are executed:
   (a) the chosen message is removed from the inbox of the card *$receiver*;
   (b) the value *$value* is removed from the balance of the card *$receiver*;
   (c) the card *$receiver* puts a VAL message in the inbox of card *$na* and in its outbox.

By means of the verification of the CTL property reported in Code 9.1, we have discovered that there is a situation in which the system is in deadlock:

```
-> State: 1.1 <-
  authentic(CARD1) = TRUE
  authentic(CARD2) = TRUE
  balance(CARD1) = 5
  balance(CARD2) = 5
  isFree(CARD1) = TRUE
  isFree(CARD2) = TRUE
  inbox(CARD1, ACK, CARD1, 0, 1) = FALSE
  ...
  inbox(CARD2, VAL, CARD2, 5, 2) = FALSE
  outboxIsNone(CARD1) = TRUE
  outboxIsNone(CARD2) = TRUE
  outboxMessage(CARD1) = MSGTYPE_UNDEF
  outboxMessage(CARD2) = MSGTYPE_UNDEF
  outboxMoney(CARD1) = -2147483647
  outboxMoney(CARD2) = -2147483647
  outboxName(CARD1) = NAME_UNDEF
  outboxName(CARD2) = NAME_UNDEF
  outboxTid(CARD1) = TIDDOM_UNDEF
  outboxTid(CARD2) = TIDDOM_UNDEF
  tids(1) = FALSE
  tids(2) = FALSE
  lv_$card_0 = CARD2
  lv_$rule_0 = STARTTORULE
  lv_$na_1 = CARD1
  lv_$value_1 = 0
  lv_$tid_1 = 1
  ...
-> Input: 1.2 <-
-> State: 1.2 <-
  isFree(CARD2) = FALSE
  inbox(CARD1, REQ, CARD2, 0, 1) = TRUE
  outboxIsNone(CARD2) = FALSE
  outboxMessage(CARD2) = REQ
  outboxMoney(CARD2) = 0
  outboxName(CARD2) = CARD1
  outboxTid(CARD2) = 1
  tids(1) = TRUE
  lv_$card_0 = CARD1
  lv_$na_1 = CARD2
  lv_$tid_1 = 2
```

Figure 9.11: Counterexample of the CTL property `ag(inbox(CARD1, REQ, CARD2, 0n, 1n) implies ef(inbox(CARD2, VAL, CARD1, 0n, 1n)))`

1. *CARD2* executes the rule *r_startTo*: it asks 0 money to *CARD1*[6]. The outbox of *CARD2*, after the rule has fired, contains the same message it has sent to *CARD1*.
2. *CARD1* executes the rule *r_startTo*: it also asks 0 money to *CARD2*. The outbox of *CARD1* contains the same message it has sent to *CARD2*.

At this point, in order to continue the transfers of money, the two cards should execute the rule *r_req* to satisfy the request of the other card. A receiver, however, in the *r_req* rule satisfies a REQ message only if its outbox is empty. But both cards have their outboxes occupied by the REQ message that they have sent to the other card. So, we have reached a situation in which the two cards are blocked.

The used CTL property is

```
ag(inbox(CARD1, REQ, CARD2, 0n, 1n) implies ef(inbox(CARD2, VAL, CARD1, 0n, 1n)))
```

The property checks that, if *CARD2* has done a request of 0 money to *CARD1*, sooner or later *CARD1* will reply with the VAL message. The property is not verified and Fig. 9.11 shows the counterexample. In State 1.1 *CARD2* has requested, in the rule *r_startTo*, 0 money to *CARD1* (the tid is 1): we can see that there is the request by observing the logic variables `lv_$card_0`, `lv_$rule_0`, `lv_$na_1`, `lv_$value_1` and `lv_$tid_1`. In State 1.2 the message has been delivered to *CARD1* (`inbox(CARD1, REQ, CARD2, 0, 1) = TRUE`) and put in the outbox of *CARD2* (`outboxIsNone(CARD2) = FALSE`, ..., `outboxTid(CARD2) = 1`).

In State 1.2, observing the same logic variables we have observed previously, we can see that *CARD1* has requested, in the rule *r_startTo*, 0 money to *CARD2* (the tid is 2)[7]. So, in the next state (State 1.3 not shown by the returned counterexample), the outbox of *CARD1* and the inbox of *CARD2* will contain this request.

From State 1.3, both cards can not reply to the request of money from the other card (although they have enough money in their balances): in fact, both the outboxes of *CARD1* and *CARD2*

[6]We can notice that the request will be always satisfied, because there will be always enough money on the balance of *CARD1* to satisfy a request of 0 money.

[7]In NuSMV counterexamples, the content of a state is given as difference with respect to the previous state.

are occupied by a message and the *r_req* rule can be executed only if the outbox of the receiver is empty.

We now present two possible solutions to fix the problem. We applied both solutions to the model shown in Code 9.1 and in both cases the CTL property has been verified.

Naïve solution In the AsmetaL model described in Section 9.2.2.1 we have not considered the ABORT rule described in [154], that permits to abort a transaction. In fact we thought that, even without the ABORT rule, the transfer of money between two cards should be always permitted if the balances of the cards allow it.

However, we have seen that the reintroduction of the ABORT rule can solve the problem, because, if the system enters in deadlock, sooner or later the execution of the ABORT rule terminates one of the two transactions, so solving the deadlock.

We do not think that this is a good solution and so we propose a modification to the model presented in [154] to fix the problem, without the need of using the ABORT rule.

Proposed solution We propose to add to the model a boolean function *check($initiator in Name)* that verifies that card *$initiator* has not a pending request of money from another card. In the rule *r_startTo*, the guard of the choose rule is extended with the control *check($initiator)*: in this way, card *$initiator* can not request money if it has to fulfil a request of money previously received from another card. This control avoids the deadlock. These are the required modifications to the model shown in Code 9.1.

```
signature:
    derived check: Prod(Name) −> Boolean

definitions:
    function check($initiator in Name) =
        not(exist $na in Name, $v in MoneyDom, $t in TidDom with inbox($initiator, REQ, $na, $v, $t))

    macro rule r_startTo($initiator in Name) =
        if( isFree( $initiator )) then
            choose $na in Name, $value in MoneyDom, $tid in TidDom with not(tids($tid)) and
                                                                  authentic($na) and $na != $initiator
                                                                  and check( $initiator ) do
                par
                    ...
                endpar
        endif
```

Chapter 10

Runtime monitoring

According to [124], *runtime monitoring* (also *runtime verification*) is “the discipline of computer science that deals with the study, development, and application of those monitoring techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property”.

So, the aim of runtime monitoring is to check that the observed executions of a system[1] ensure some correctness properties. Runtime monitoring is a *lightweight* verification technique that, considering the ability to detects faults, can be classified halfway between those techniques that try to ensure universal correctness of systems – as model checking and theorem proving (see Chapter 9) – and those techniques like testing that ensure the correctness only for a fixed set of executions (i.e., those deriving from the test cases specified in the test suite).

The main difference with techniques like model checking is that, whereas these techniques check all possible executions of a program, runtime monitoring only checks those executions that are actually performed by the program under scrutiny. So, it is possible that, although the program contains a fault, its executions never produce a failure that evidences that fault.

The main difference with testing, instead, is that the number of executions over which the program is checked is not fixed. Sometime, runtime monitoring is seen as the process of testing the system *forever* [124], since, as in testing, the actual output is checked with respect to an expected output (usually described by an *oracle*), but, unlike testing, every execution of the system is checked.

Finally, what distinguishes runtime monitoring from any other validation and verification (V&V) activity, is that it can be executed also after the deployment of the program, whereas traditional V&V activities are only executed *offline*, that is before the deployment.

So, why and when should we use a runtime monitoring technique? Let’s try to list some motivations:

- Exhaustive verification techniques are not always applicable; for example, proving a security property that a system never reaches a dangerous state could require too much time depending on the size of the system. Moreover, techniques like model checking usually check the model of the system, not the actual implementation: so, it remains the problem of verifying that the correctness of the model implies the correctness of the implementation.

- The system could strongly depend on the environment in which it is executed [53]. If this environment is not available at testing time or, although available, it is not practically possible to interact with it (because maybe too much time consuming), testing the system could become difficult. In unit testing this problem is sometimes mitigated by using *mock objects* that mimic the behaviour of the environment: however, if the actions of the environment are not fully predictable, also using mock objects could be not useful.

[1] All the definitions are applicable both to hardware and software systems. However, from now on, we will only consider software systems.

- Finally, safety-critical systems [115] as medical devices, aircraft flight controls, nuclear systems, etc., although tested and verified deeply, could require an additional degree of confidence that they behave as expected. Runtime monitoring here acts as a double check that everything goes well [124].

In Section 10.1 we give general definitions about runtime monitoring, while in Section 10.2 we give a non-exhaustive overview of the literature about runtime monitoring by describing four runtime monitoring techniques. In Chapter 11 we introduce the approach we propose to execute runtime monitoring of Java programs using ASMs. In Chapter 12 we describe how our runtime monitoring approach can be used in combination with model-based testing for testing nondeterministic systems.

10.1 Runtime monitoring schema

The general schema of a runtime monitoring approach is depicted in Fig. 10.1. We have adapted to our purposes the schema presented in [57].

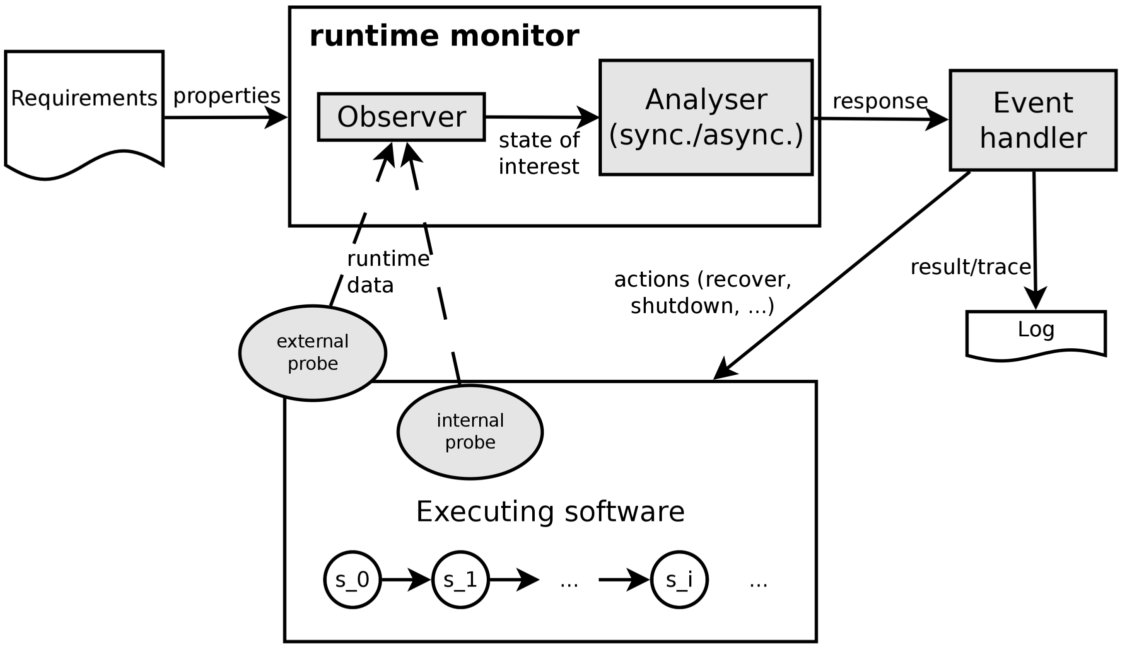


Figure 10.1: General overview of a runtime monitoring scenario

Any runtime monitoring technique must start considering the requirements that specify the expected behaviour of the software system. Some approaches are more tailored to the verification of functional requirements, whereas others address in particular non-functional requirements: here we are mainly interested in analysing those techniques that deal with functional requirements.

Starting from the requirements, a set of correctness properties [57] must be derived. These properties specify all admissible individual executions of a system and can be expressed using a great variety of different formalisms as, for example, temporal logics [92, 22], extended regular expressions [43] and Z specifications [126].

Then, these properties are usually encoded into a *runtime software-fault monitor*, or simply a *monitor*, a system that observes and analyses the states of an executing software system. The monitor checks the correctness of the system behaviour by comparing the *observed* state of the system with the *expected* state described by the correctness properties. Some approaches provide special algorithms to synthesize the monitor starting from the properties. Other approaches, instead, provide a *general* monitor that can check any property and simply needs to be initialised with the properties to check.

A monitor is composed of two elements: the *observer* that concretely monitors the program execution and the *analyser* that checks that the correctness properties are satisfied.

In order to monitor the program, the observer uses some *probes* that can collect different kinds of data about the program execution as, for example, the values of internal fields/variables, which methods have been executed, the value returned by a method, etc.. These probes can be *internal*, if they have access to the internal state of the program, or *external*, if they can just observe its external behaviour.

The runtime monitoring process works as follows:

1. When a *monitored event* is detected, i.e., a given condition is satisfied (e.g., a method has been called, or a field has been updated), the observer retrieves *runtime data* from the probes;
2. the observer extracts from the runtime data the *state of interest*, i.e., those data that are necessary for checking the properties, and sends it to the analyser;
3. the analyser, based on the received data, checks if the execution of the system still satisfies the properties. The analyser can operate in two ways:
    - *synchronously*, if the program can continue its execution only after the analyser has finished its computation; this scenario is quite intrusive and so it would be desirable that the time taken by the verification of the properties is narrowed as much as possible. However, in this scenario all the faults are captured timely, as soon as they occur.
    - *asynchronously*, if the program, in order to continue its computation, does not have to wait for the analyser to finish the checking of the properties. This approach is less intrusive of the previous one, but it could lead to situations in which a fault is discovered after it has already produced a failure in the program.
4. the analyser sends the *response* of its control to the *event handler* that, in case of property violation, can decide to make some actions on the running program (e.g., stopping it or restoring it to a previous state known to be safe) and/or record some information in a log file, maybe in the form of execution traces (similar to counterexamples of model checking).

The approach we have presented, in which the system is checked while it is executing, is sometimes called *online monitoring* [22], to distinguish it from *offline monitoring*, in which the system executions are recorded and checked offline.

10.2 Runtime monitoring techniques

In the following, we give a non-exhaustive overview of the literature about runtime monitoring by describing four runtime monitoring techniques.

10.2.1 LTL$_3$

In [22, 21] the authors presents an approach to runtime monitoring in which traces of programs are examined in order to check if they satisfy some temporal properties expressed in LTL$_3$, a linear-time temporal logic designed for runtime monitoring. The syntax of LTL$_3$ is the same as LTL (see Section 9.1.2.1); its semantic, instead, is adapted in order to handle finite traces. It uses three truth values: *true*, *false* and *inconclusive* ($\top$, $\bot$ and ?). Given a finite trace u and an LTL$_3$ formula φ, the value of φ is

a) *true* if every continuation of u satisfies φ,

b) *false* if no continuation of u satisfies φ,

c) *inconclusive* otherwise.

From an LTL$_3$ formula they derive an FSM whose states are labeled with three output symbols ($\top$, $\bot$ and ?) that constitute a classification of the finite traces of the monitored system.

During the runtime monitoring, the FSM is visited using as inputs the events observed in the running software. The visit continues as long as the visited states are labeled with ?, since this means that no conclusive result can be given about the formula. As soon as a state labeled with $\top$ or $\bot$ is reached, the verification is interrupted because a conclusive result has been reached and no continuation of the observed trace can modify the result.

Let's see the example presented in [22]. In C++ a known problem is the *static initialisation order fiasco* [58], that is caused by the fact the initialisation of static objects is executed in a nondeterministic order. So, if threads get spawned before the `main` method is executed, it could be that not all the resources necessary to synchronize those threads are already initialised. The authors propose the following simple LTL formula to monitor the execution of a C++ program:

$$\varphi \equiv \neg spawn\ \mathbf{U}\ init$$

where the atomic proposition *spawn* signals if a thread has been spawned, while *init* signals if the initialisation of the program has finished. Property φ simply requires that no thread can be spawned until the application has not finished its initialisation. Fig. 10.2 shows the Büchi automaton that describe φ, i.e., that accepts all the infinite traces that are models for φ.

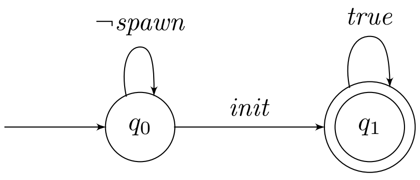


Figure 10.2: Büchi automaton that describe property $\varphi \equiv \neg spawn\ \mathbf{U}\ init$

Fig. 10.3 shows the FSM used, during runtime monitoring, as monitor for property φ.

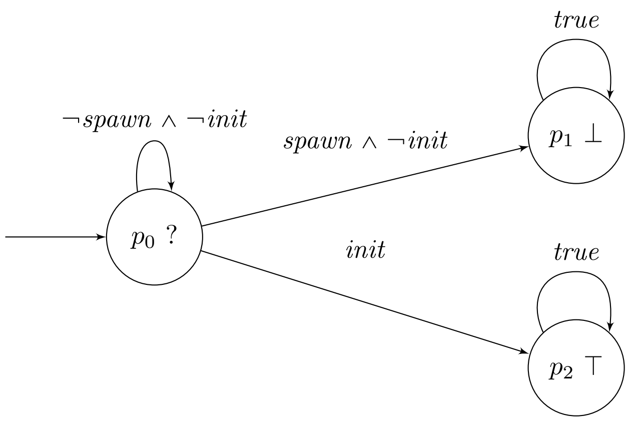


Figure 10.3: Monitor for property $\varphi \equiv \neg spawn\ \mathbf{U}\ init$

As long as the monitor does not read as input *spawn* or *init*, it remains in state p_0, in which no conclusive result can be established. If the monitor reads an *init*, it goes in state p_2 where the property is declared *satisfied*. Instead, if it reads a *spawn* without an *init*, it goes to state p_1

where the property is declared *violated.* Any trace continuing from p_1 or p_2 can not change the result about the property and, so, the monitoring can be interrupted.

10.2.2 Monitored oriented programming

Monitored-oriented programming (MOP) [43, 44] is a generic monitoring framework whose aim is the integration of an implementation with its formal specification, by checking the latter against the former at runtime. The specification can be written in any formalism for which a logic plugin has been developed: the framework yet implements several logic plugins for different formalisms as FSMs, Extended Regular Expression (ERE), LTL, and Past Time LTL (PTLTL).

According to the authors [43], MOP can be seen as a *lightweight formal method* that extends *programming languages with logics*: logic statements can be placed in different places of the program for monitoring its execution and, in case of errors, also possibly undertaking some recovery actions.

The approach requires that the formal specifications are translated (in two steps) in the target programming language. The obtained monitoring code can be used in online mode, in which the monitoring code is placed in the monitored program, and in an offline mode in which it is used to check traces recorded by adequate probes. Aspect Oriented Programming (AOP) [113] is used to weave the monitoring code into the monitored code.

The MOP approach has been instantiated in JavaMOP [45], for the monitoring of Java programs, and in BusMOP [147], for the monitoring of PCI bus traffic.

Let's consider JavaMOP. JavaMOP already provides several logic plugins; on the project website [103] there are several specifications that can be downloaded or translated into a monitor using an online tool (also the offline version is available). The obtained monitor is an AspectJ [132] aspect that is responsible for catching the monitored events and executing the checking. Let's take an LTL specification from their web repository to describe the approach. Code 10.1 shows the JavaMOP LTL specification that states that, when using an `Iterator`, you always have to call the method `hasNext` before calling the method `next`. Such specification describes a common requirement for a *safe* usage of iterators.

```
package mop;
import java.io.*;
import java.util.*;

HasNext(Iterator i) {
    event hasNextTrue after(Iterator i) returning(boolean b): call(* Iterator.hasNext())
            && target(i) && condition(b) { }
    event next before(Iterator i) : call(* Iterator.next()) && target(i) { }

    ltl: [](next => (*) hasNextTrue)

    @violation {
        System.out.println("Method next has been called without calling hasNext before!");
    }
}
```

Code 10.1: JavaMOP LTL specification: correct usage of iterators

In the specification, the atomic predicates used in the LTL formula must be declared using the keyword *event.* An event looks like an AspectJ pointcut, that is a point of the program execution one wants to capture. In the example, the event *hasNextTrue* captures the points of the program execution just after calls of the method `hasNext` that has returned the value *true.* The event *next*, instead, identifies the points of the program execution just before calls of the method `next`.

The LTL formula states that always ([]), if the method `next` is called, previously[2] ((*)) the

[2]Here *previously* indicates the event occurred right before the current event.

method `hasNext` must have been called returning *true*.

The section `@violation` permits to specify the code that must be executed if the property is violated. In this case, we simply require that a message is printed to the standard output. However, any action could be made: e.g., calling the method `hasNext` to check if it is possible to proceed with the execution of method `next` anyway[3] and, if not, doing some recovery actions, i.e.,

```
@violation {
    if(!i.hasNext()) {
        //do some recovery actions
    }
}
```

In order to be used for monitoring, the specification in Code 10.1 must be translated into a monitor, that is an AspectJ aspect. We do not report here the obtained aspect because too big and not really interesting. We want to show, instead, what is the effect of the monitoring.

Let's consider Code 10.2 in which an iterator over a list of strings is created.

```
public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<String>();
    list.add("a");
    list.add("b");
    list.add("c");
    Iterator<String> it = list.iterator();
```

Code 10.2: Creation of an iterator for a list of strings

In the following we will see some instructions that could be executed after those shown in Code 10.2.

Fig. 10.4 shows, on the left, a correct usage of the iterator in which the execution of `next` is always guarded by an execution of `hasNext`. On the right the produced output is reported. Since no violation of the LTL property occurs, the execution ends correctly.

```
while(it.hasNext())
    System.out.println(it.next());
```

```
a
b
c
```

Figure 10.4: Correct usage of the iterator

Fig. 10.5, instead, shows a wrong usage of the iterator, since the method `next` is called without calling `hasNext` before. The monitor detects the violation and executes section `@violation`. Note that, however, the violation of the property does not imply a failure of the Java code because, in this case, there is an element that can be retrieved with `next`.

```
System.out.println(it.next());
```

```
Method next has been called without calling hasNext before!
a
```

Figure 10.5: Wrong usage of the iterator – No Java exception risen

Also Fig. 10.6 shows a wrong usage of the iterator. In this case, however, also a failure in the Java code occurs (exception `NoSuchElementException` is risen), since, when the `next` method is called outside the while loop, there is no element in the iterator that can be retrieved.

[3]Note that the event *next* captures the execution point right before the call of method `next`. So, if in section `@violation` method `hasNext` is called, it correctly states if there is an available element in the iterator, to be consumed by the captured call of method `next`.

```
while(it.hasNext())
  System.out.println(it.next());
System.out.println(it.next());
```

```
a
b
c
method next has been called without calling hasNext before!
Exception in thread "main" java.util.NoSuchElementException
        at java.util.ArrayList$Itr.next(ArrayList.java:757)
        at mop.Main.main(Main.java:16)
```

Figure 10.6: Wrong usage of the iterator – Java exception risen

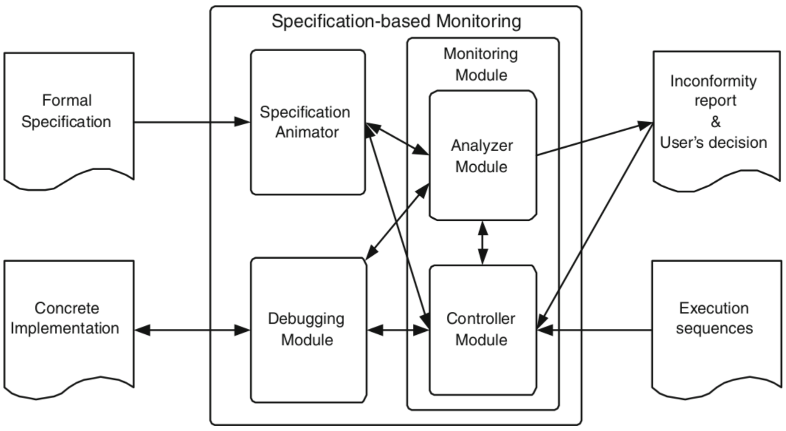


Figure 10.7: Software monitoring through Z specifications

10.2.3 Software monitoring through Z specifications

In [126] a *formal specification-based software monitoring system* is presented. In this approach the behaviour of a concrete implementation (a Java code) is checked for compliance with a formal specification.

Unlike MOP, in this approach the concrete implementation is separated from the specification and no instrumentation code is inserted in the monitored program.

Fig. 10.7 shows the schema of the approach, as reported in [126]. The execution of program is monitored by a *debugger* and the formal specification is executed in parallel with a *specification animator*. The user must provide to the system an *execution sequence*, i.e., a list of the methods that must be executed, together with values to be used as actual parameters. The *controller module* drives the execution according to the execution sequence. The *analyser module* compares the execution of the program and the simulation of the formal specification to see if they are conformant: in case of violation, it reports the error to the user.

In this approach, the Z specification language [170] is used to write the formal specifications. Z is a formal language, based on set theory and first order logic, for specifying sw/hw systems; a system is modeled describing its states and the way in which they can be modified. A specification contains *state schemas* and *operation schemas*. A state schema contains variable declarations and related invariants: the system state is given by the values of the variables that must always respect the invariants. An operation schema describes the relation between the states before and after an operation execution.

Fig. 10.8 shows the Z specification of a FIFO queue. The state schema *Queue* describes the queue as a sequence of natural numbers that can contain, at the most, ten elements. Operation schema *InitQueue* describes the initialisation of the queue, *Enqueue* the addition of an element to the queue, and *Dequeue* the removal of an element.

$size == 10$

Queue

$items : \mathrm{seq}\,\mathbb{N}$

$\#items \leq size$

InitQueue

$Queue'$

$items' = \langle\rangle$

Enqueue

$\Delta Queue$

$item? : \mathbb{N}$

$\#items < size$

$items' = items \frown \langle item? \rangle$

Dequeue

$\Delta Queue$

$item! : \mathbb{N}$

$\#items > 0$

$item! = head\ items$

$items' = tail\ items$

Figure 10.8: Z formal specification of a FIFO queue

The actual monitoring of the implementation must be done through a tool in which the user must specify the sequence of methods she wants to execute and where she can follow the parallel execution of the program and of the specification. Note that the proposed approach can be used during the testing of a program, but can not be used in the deployed program in which the monitoring system should be hidden to the final user (as in JavaMOP, for example). Fig. 10.9 shows a screenshot of the tool in which the Java implementation of a FIFO queue has been monitored with the specification in Fig. 10.8: since the Java code wrongly implements method `deQueue` (it removes the last inserted element) a conformance violation is risen.

10.2.4 Dynamic Monitoring – Dynamo

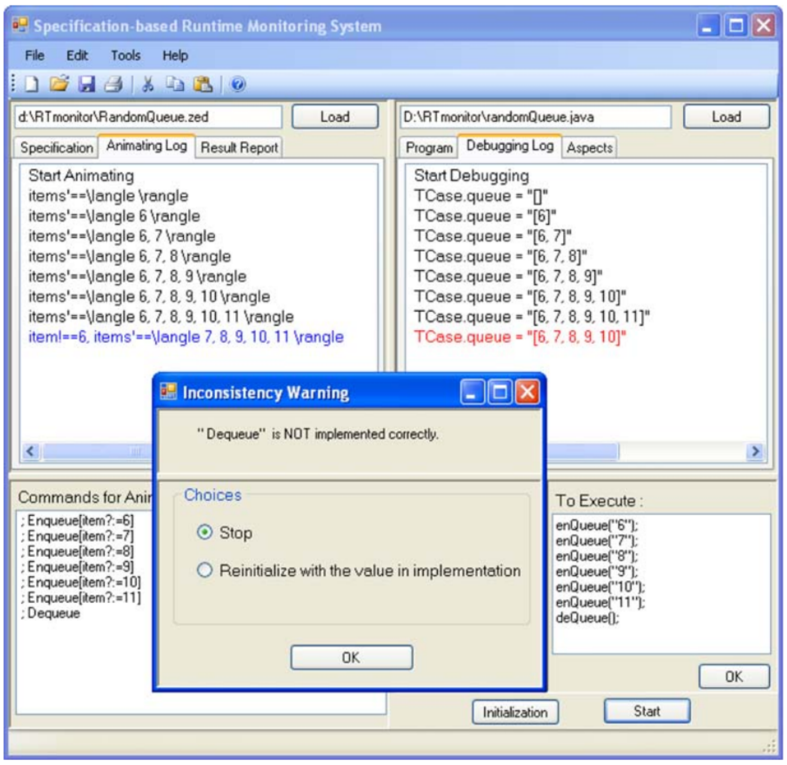
Business Process Execution Language (BPEL) is a language used to execute web services orchestrations. A BPEL specification describes the evolution of a business process specifying a sequence of activities; a BPEL engine executes a BPEL specification, performing all the activities and finding the required external services. The binding between the external service invocation and the actual service exported by a service provider is done at runtime and can change over time [83] (e.g., new versions of the selected services are released, services are supplied by different providers, . . .). In such a scenario, where the binding with web services is made and can change dynamically, the typical V&V activities are less effective than in the traditional scenario. In the *web service orchestration* scenario, instead, a continuous verification should be executed, for example, that the services delivered comply with the requests.

In [18] the tool *Dynamo* (Dynamic Monitoring) is presented: it permits to execute runtime monitoring of BPEL processes. It can check that both functional and non-functional properties are satisfied and, eventually, recover from erroneous situations.

Dynamo has a language, called *WSCoL* [19], that permits to specify constraints on WS-BPEL processes: it permits to specify pre-conditions and post-conditions for the BPEL activities that use external services: *invoke* (the process invokes a web service), *receive* (the process receives a message after the invocation of a web service), *reply* (the process returns a message to the partner that started the conversation) and *pick* (the process selects an activity associated with the message received). In Fig. 10.10 an overview of the Dynamo system is shown.

Let's see how to use Dynamo in practice:

1. First of all, we must write a WS-BPEL specification; in Fig. 10.10 *WS-BPEL process* contains the invocations of the three services *A*, *B* and *C*. It is important to notice that this specification does not contain any WSCoL code (this specification is also called the *unmonitored version* of the business process) and it can be run by the BPEL engine: on the left side of the figure we can see that the normal run leads to the invocation of web services *WS A*, *WS B* and *WS C*.



d:\RTmonitor\RandomQueue.zed

D:\RTmonitor\randomQueue.java

```
Start Animating
items'==\langle \rangle
items'==\langle 6 \rangle
items'==\langle 6, 7 \rangle
items'==\langle 6, 7, 8 \rangle
items'==\langle 6, 7, 8, 9 \rangle
items'==\langle 6, 7, 8, 9, 10 \rangle
items'==\langle 6, 7, 8, 9, 10, 11 \rangle
item!==6, items'==\langle 7, 8, 9, 10, 11 \rangle
```

```
Start Debugging
TCase.queue = "[]"
TCase.queue = "[6]"
TCase.queue = "[6, 7]"
TCase.queue = "[6, 7, 8]"
TCase.queue = "[6, 7, 8, 9]"
TCase.queue = "[6, 7, 8, 9, 10]"
TCase.queue = "[6, 7, 8, 9, 10, 11]"
TCase.queue = "[6, 7, 8, 9, 10]"
```

" Dequeue" is NOT implemented correctly.

[✓ Stop]
Reinitialize with the value in implementation

```
; Enqueue[item?:=6]
; Enqueue[item?:=7]
; Enqueue[item?:=8]
; Enqueue[item?:=9]
; Enqueue[item?:=10]
; Enqueue[item?:=11]
; Dequeue
```

```
enQueue("6");
enQueue("7");
enQueue("8");
enQueue("9");
enQueue("10");
enQueue("11");
deQueue();
```

Figure 10.9: Tool for runtime monitoring through Z formal specifications

2. Then, the constraints that we want to check must be specified in the *Monitoring Definition File*, using the WSCoL language; in this file we have also to specify the BPEL activities that are monitored by the constraints. In the example, the *Monitoring Definition File* specifies some constraints on the invocation of service *B*.

3. The component BPEL^2 takes in input the *WS-BPEL Process* and the *Monitoring Definition File*; it builds the monitored version *Instrumented WS-BPEL process* where each BPEL invocation that is monitored by a constraint (as specified in the *Monitoring Definition File*) is replaced by an invocation of the *Monitoring Manager MM*. Moreover, BPEL^2 adds, at the beginning of the specification, some code to set up the monitoring manager before the process execution, and some code at the end to release it at the completion of the process execution.

4. During the run of *Instrumented WS-BPEL process* (right side of Fig. 10.10) we have two kinds of invocations:

   - *non-monitored invocations* that are dealt like in a run of *WS-BPEL Process*: invocations of non-monitored services *A* and *C* lead to the invocations of web services *WS A* and *WS C*;
   - *monitored invocations* that are handled by the *Monitoring Manager MM*.

5. The *Monitoring Manager MM*, for each monitored invocation, executes the following tasks:

   - it acts as a proxy, invoking the suitable web service for the current invocation;
   - it checks, with a proper *data analyser*, that the constraints on the current invocation are satisfied. This check must be executed before or after the web service invocation,

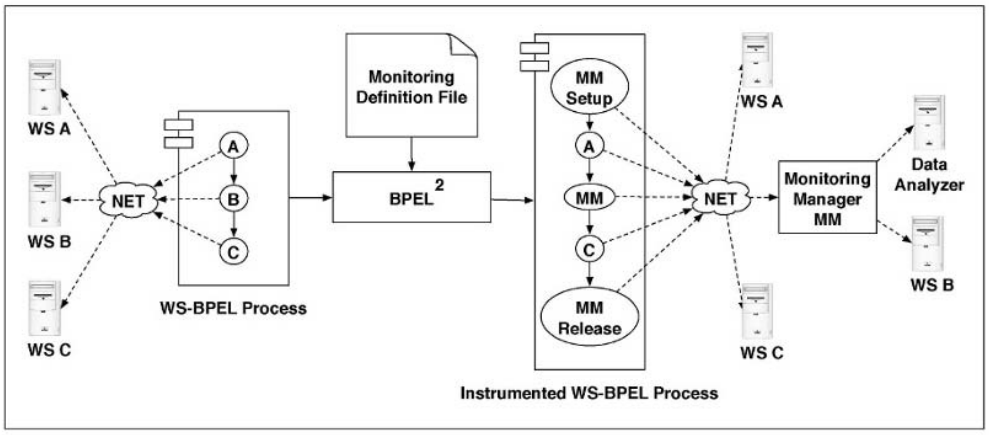


Figure 10.10: Dynamo – Dynamic Monitoring

depending on the fact that the constraints are, respectively, pre or post conditions. It is possible to use different data analysers and, so, to analyse different kinds of properties.

Since this is an online monitoring approach (see Section 10.1), all the activities of the *Monitoring Manager MM* are blocking, that is they block the execution of the business process during the checking of the constraints: this permits to discover erroneous behaviours as soon as they occur. When a violation is detected, Dynamo can also execute some recovery actions, that must be specified in the *Monitoring Definition File.*

Chapter 11

Runtime monitoring through ASMs

In Section 10.2 we have seen that, in most of the approaches dealing with runtime monitoring of software, the required behaviour of the system is formalized by means of correctness properties [57] described using *declarative* specifications, as LTL/PTLTL [22, 108, 45], extended regular expressions [45], design by contract languages [19, 122], and so forth. Also new notations developed for runtime monitoring are based on a declarative style. ALBERT [17] is a temporal assertion language for the runtime monitoring of BPEL processes that extends LTL with timestamps. Tracematches [1], instead, are an extension of AspectJ pointcuts, based on the integration of regular expressions, that can be used with success in runtime monitoring: indeed, while standard pointcuts only permit to capture single *point*s of the program execution, tracematches permit to signal if particular *run*s of the program have been executed, a feature requested by several runtime monitoring techniques.

Declarative notations have the advantage of having been extensively used in traditional verification techniques and, moreover, they provide efficient algorithms to derive the monitors that must be used during the runtime checking. For these reasons they have been widely used in runtime monitoring.

The use of *operational* specifications (as abstract automata and state machines) for runtime monitoring, instead, has not been studied with the same strength. An operational specification describes the desired system behaviour by providing a model implementation of the system, generally executable. The work in [126], that we have briefly presented in Section 10.2.3, is an example of runtime monitoring that makes use of operational specifications: the formal specification of the program under monitoring is given in the Z language and it describes the system state and the ways in which it changes. Another approach that uses operational specifications (called *model programs*) is presented in [20], where they use ASMs to specify all of the traditional design-by-contract concepts of pre- and post-conditions and invariants for .NET components. The work we propose in this chapter has been inspired by both works [126, 20].

We claim that specification styles (and languages) may differ in their expressiveness and very often their use depends on the preference and taste of the specifier, the availability of supporting tools, and so forth. Up to now, declarative languages have been preferred for runtime software monitoring, but we think that the use of operational languages should be investigated more deeply.

In this chapter we present *CoMA* (Conformance Monitoring by ASMs), a specification-based approach and its supporting tool for runtime monitoring of Java software. We assume that the desired system behaviour of a Java program under monitoring is given in an *operational* way by means of an ASM (see Section 2.1).

The technique we propose makes use of Java annotations. However, annotations do not contain the specification of the correct behaviour (like in JML [122] or in MOP (see Section 10.2.2)) but they are only used to link the concrete implementation to its formal model, keeping separated the implementation of the system and its high-level specification. The approach has, therefore, the advantage of allowing the reuse of abstract formal specifications for other purposes

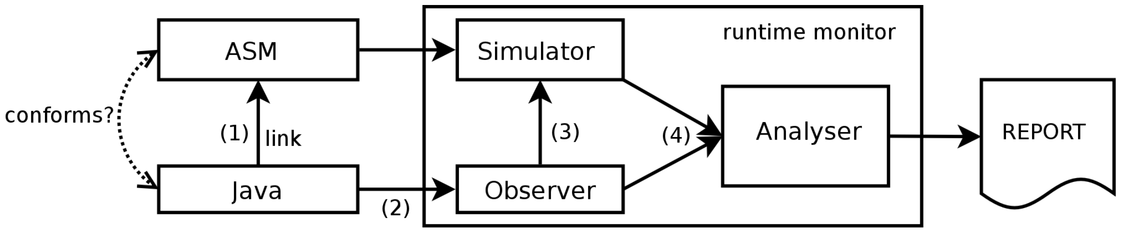


Figure 11.1: The CoMA runtime monitor for Java

(as also discussed in [20]), like formal verification, model simulation, model-based testing, and so forth.

In Section 11.1, we present the theoretical framework of *CoMA*, in which we explain the relationship between the Java implementation and its ASM specification. This relationship defines syntactical links or mappings between Java and ASM elements and a semantical relation which represents the conformance. In Section 11.2, we introduce the actual implementation of our conformance monitoring approach which is based on Java annotations and AspectJ. In Section 11.3, we discuss some advantages and limits of our approach; by means of diverse examples, we evaluate performance, expressiveness and usability of CoMA w.r.t. other approaches for runtime monitoring. Finally, in Section 11.4, we describe how our approach can be used for the runtime monitoring of web services.

11.1 Runtime conformance monitoring based on ASMs

In our approach we intend runtime monitoring as conformance analysis at runtime and we propose *CoMA*, runtime *Conformance Monitoring* of Java code *by ASM specifications*.

The CoMA monitor supports *online* monitoring, namely it considers executions in an incremental fashion. It takes as input an executing Java program and an ASM formal model. The monitor observes the behaviour of the Java program and determines its correctness w.r.t. the ASM specification working as an oracle of the expected behaviour. While the software system is executing, the monitor checks conformance between the observed state and the expected state.

We now instantiate the general schema of a runtime monitoring technique, presented in Section 10.1, to our approach, as depicted in Fig. 11.1.

A *link* between a Java class and an ASM must be provided ((1) in Fig. 11.1) in order to describe the conformance relation; a set of annotations is used to this purpose. The monitor is composed of: an *observer* that evaluates when the Java (observed) state is changed ((2) in the figure), and leads the ASM to perform a machine step ((3) in the figure), and an *analyser* that evaluates the step conformance between the Java execution and the ASM behaviour ((4) in the figure). When the monitor detects a violation of conformance, it reports the error. It can also produce a trace in form of counterexample, which may be useful for debugging. Note that the use of CoMA can be twofold: also faults in the specification can be discovered by monitoring software. For instance, by analysing and re-executing counterexamples, faults in the model can be exposed.

In the following sections, we introduce the theoretical basis of our monitoring system. We, therefore, formally define what is an observed Java state, how to establish a conformance relation between Java and ASM states and, therefore, step conformance and runtime conformance between Java and ASM executions.

11.1.1 Observable Java elements

In order to mathematically represent a class and the state of its objects, we introduce the following definitions.

Definition 11.1 (Class). *A* class *C is a tuple $\langle c, f, m\rangle$ where c denotes the non-empty set of constructors, f is the set of all the fields, m is the set of methods.*

We denote the public fields of C as f^{pub} while the public methods are denoted as m^{pub}. Among the methods of a class, we distinguish also the *pure* methods:

Definition 11.2 (Pure method). Pure method*s m_{pure} are side effect free, with respect to the object/program state. They return a value but do not assign values to fields. m^{pub}_{pure} denotes the set of all pure public methods in m.*

Pure methods [54] are useful and common specification constructs. By marking a method as pure, the specifier indicates that it can be treated as a function of the state (as in JML [122]). We consider only pure methods without arguments.

Definition 11.3 (Virtual state). *Given a class $C = \langle c, f, m\rangle$, the* virtual state*, $VS(C)$, is given by $VS(C) = f^{pub} \cup m^{pub}_{pure}$.*

Definition 11.4 (Observed state). *We define* observed state*, $OS(C) \subseteq VS(C)$, as the subset of the virtual state consisting of all public fields, and pure public methods of the class C the user wants to observe.*

Therefore, $OS(C)$ is the set of Java elements monitored at runtime. For convenience, we can see $OS(C) = OF(C) \cup OM(C)$ to distinguish between the subset of *observed fields* $OF(C)$ and the subset of *observed methods* $OM(C)$ of $OS(C)$. Note that $OF(C) \subseteq f^{pub}$ and $OM(C) \subseteq m^{pub}_{pure}$. The (returned) values of the elements of $OS(C)$ can change by executing any not pure method (in $m_{\neg pure} = m - m_{pure}$).

Definition 11.5 (Changing method). *Given a Java class C, we define* changing methods*, $changingMethods(C) \subseteq m_{\neg pure}$, all methods of C whose execution is responsible for changing an element of $OS(C)$ and that the user wants to observe.*

Definition 11.6 (Observed constructor). *We define* observed constructor*s, $OC(C) \subseteq c$, the set of constructors whose execution the user wants to monitor.*

Since the observed constructors represent the points of the program execution in which the monitoring must start, we require that $OC(C) \neq \varnothing$.

Definition 11.7 (Observed input). *We define* observed input*s, $OI(C)$, all the formal parameters of methods in $OC(C) \cup changingMethods(C)$ that the user wants to monitor.*

11.1.2 Link between the Java program and the ASM

Linking observable Java elements to ASM entities In order to be runtime monitored, a Java class C should have a corresponding ASM model, ASM_C, abstractly specifying the behaviour of an instance of the class C.

Observable elements of the observed state of class C must be linked to the controlled functions $ContrFuncs(ASM_C)$ of the ASM model ASM_C. The function

$$linkOS : OS(C) \rightarrow ContrFuncs(ASM_C) \tag{11.1}$$

yields the set of the ASM controlled functions[1] linked to the observable Java elements of C. The function *linkOS* is not surjective because there are ASM controlled functions that are not used in the conformance analysis. The function in neither injective, since different elements of $OS(C)$ can be linked to the same ASM function.

[1]CoMA also permits to link elements of $OS(C)$ to locations, rather than functions: in order to keep the explanation as simple as possible, we do not include this feature in the formal definitions. However, all the definitions given in terms of functions can be adapted to locations.

The observed inputs $OI(C)$ must be linked to the monitored functions $\mathit{MonFuncs}(ASM_C)$ of the ASM model. The function

$$\mathit{linkOI} : OI(C) \rightarrow \mathit{MonFuncs}(ASM_C) \tag{11.2}$$

establishes the link. Monitored functions are suitable to represent the constructors/methods parameters because, in an ASM, they represent the part of the dynamic state that is determined by the external environment and not by the machine, as the actual parameters do in a program. The function *linkOI* must be bijective because each monitored function must have a corresponding parameter in the Java code[2] (i.e., surjective) and, moreover, different elements of $OI(C)$ must be linked to different functions (i.e., injective).

Execution step in Java and in the ASM In order to define a step of a Java class execution, we rely on the concept of *machine step* and *last state* of execution sequence defined in the Unifying Theories of Programming (UTP) [97]. A Java *state* of an instance of a class C is the set of the actual values of its fields.

Definition 11.8 (Java Step). *Let m be a method of a Java class. A Java step is defined as the relation (s, m, s') where s is the starting state of the execution of m and s' the last state of this execution.*

Definition 11.9 (Change Step). *Let C be a Java class. A* change step *is defined as a Java step for $m \in \mathit{changingMethods}(C)$.*

Note that, choosing the granularity of the Java step at the level of class method and not at the level of single assignment, allows the designer to tune the desired granularity of the monitoring.

ASM *state* and ASM computation *step* have been defined in Section 2.1.

11.1.3 State conformance, step conformance and runtime conformance

We now formally relate the execution of a Java class instance with a simulation of the corresponding ASM model.

In the following definitions, let C be a Java class, O_C any instance of C, and ASM_C its corresponding ASM abstract model.

We assume that the function $val_{Java}(e, s)$ yields the value of a Java element $e \in VS(C)$ of C in a given state s of O_C, while the value of an ASM function a in a state S is given by $val_{ASM}(a, S)$[3]. Moreover we assume that there exists a conformance $\overset{conf}{=}$ relation among Java and ASM values [8].

Definition 11.10 (State Conformance). *We say that* a state *s of O_C* conforms to a state *S of ASM_C if all observed elements of C have values in O_C conforming to the values of the functions in ASM_C linked to them, i.e.,*

$$\mathit{conf}(s, S) \equiv \forall e \in OS(C) : val_{Java}(e, s) \overset{conf}{=} val_{ASM}(\mathit{linkOS}(e), S) \tag{11.3}$$

Definition 11.11 (Step Conformance). *We say that a* change step *(s, m, s') of an instance O_C (i.e., $m \in \mathit{changingMethods}(C)$)* conforms with a step *(S, S') of ASM_C if $\mathit{conf}(s, S) \wedge \mathit{conf}(s', S')$.*

$$\begin{tikzcd}
ASM_C & S \arrow[r, "simulation\ step"] & S' \\
O_C & s \arrow[u, dotted, "conf"] \arrow[r, "invocation\ of\ method\ m"] & s' \arrow[u, dotted, "conf"]
\end{tikzcd}$$

[2]Actually, in Section 11.2.1 we will see that it is possible to add another monitored function (not linked through *linkOI*) in the ASM model, to deal with a particular kind of nondeterminism of the Java code. However, in the current definition, we can avoid considering that particular situation.

[3]The value of an ASM function is *composed* by the values of its locations. So, in order to have a finite number of locations, we require that the function is either 0-ary or that the cardinality of its domain is finite.

$$\begin{tikzcd}
ASM_C & {} \arrow[r, "init"] & S_0 \arrow[r, dotted] & S_j \arrow[r, "step"] & S_{j+1} \arrow[rr, "step"] & & {} \\
O_C & {} \arrow[r, "inst"] & s_0 \arrow[r, dotted] \arrow[u, dotted] & s_k \arrow[r, "CM"] \arrow[u, dotted] & s_{k+1} \arrow[r, squiggly, "notCM^*"] \arrow[u, dotted] & s_{k+2} \arrow[r, "CM"] \arrow[ul, dotted] & {}
\end{tikzcd}$$

Figure 11.2: Runtime conformance

Definition 11.12 (Runtime Conformance)**.** *Given an* observed computation *of a Java instance O_C, we say that C is* runtime conforming *to its specification ASM_C if the following conditions hold:*

1. *the initial state s_0 of the computation of O_C* conforms *to the initial state S_0 of the computation of ASM_C, i.e., it yields $conf(s_0, S_0)$;*
2. *every observed change step (s, m, s') with s the current state of O_C,* conforms *with the step (S, S') of ASM_C with S the current state of ASM_C;*
3. *no specification invariant of ASM_C is ever violated.*

Fig. 11.2 depicts the co-simulation of an instance O_C and its specification ASM_C. Def. 11.12 requires conformance between s_0 and S_0. If O_C is in state s_k, executes a changing method CM, and moves to state s_{k+1}, then s_k must conform to the current ASM state S_j and s_{k+1} must conform to the next ASM state S_{j+1}. Then, no conformance check is performed until the next observed state s_{k+2} when a changing method is invoked again. Note that the final state of a Java change step and the initial state of the subsequent change step are both *state conforming* to the same abstract state of the ASM (e.g., s_{k+1} and s_{k+2} are both state conforming to S_{j+1}).

11.1.3.1 Runtime conformance in the presence of nondeterminism

Definition 11.12 assumes that, in any computation, the next state of a Java class instance O_C and of its specification ASM_C are unique. Thus, the definition is adequate for deterministic systems in which the nondeterminism is *external*, i.e., it is limited to monitored (external) quantities (e.g., which method has been called or what values have been used as actual parameters). Once these quantities are fixed by the environment, the evolution of the system is, however, deterministic.

For dealing with *internal* nondeterminism (e.g., a method behaves nondeterministically), we have to extend our conceptual framework. From now on we always intend nondeterminism as internal.

We have identified the following two scenarios:

- Nondeterministic Java class and nondeterministic ASM specification. A class method has nondeterministic behaviour (for instance it contains a call to a method in the class `java.util.Random`), and so also the abstract specification.
- Deterministic Java class and nondeterministic ASM specification. This situation arises when the ASM model is more abstract (with less implementation details) than the corresponding Java code.

In case a class C or its model ASM_C are nondeterministic, the next computational state of O_C or ASM_C is not always uniquely determined and, therefore, their conformance, according to Def. 11.12, may fail not because of a non-conformant behaviour of the implementation, but because O_C and ASM_C may choose two next states which are not conformant. We here refine points 1 and 2 of Def. 11.12 of runtime conformance for dealing with nondeterminism, distinguishing between *weak* and *strong* conformance. For the *weak* conformance, we require that the next step of O_C is state-conforming with *at least one* of the next states of the specification ASM_C. For the *strong* conformance, we require that the next step of O_C is state-conforming with *one and only one* of the next states of the specification.

Definition 11.13 (Weak [Strong] runtime conformance)**.** *We say that C is* weakly [strongly] *runtime conforming to its specification ASM_C if the following conditions hold:*

1. *the initial state s_0 of the computation of O_C conforms to* at least one [one and only one] *initial state S_0 of the computation of ASM_C, i.e., $\exists$ [$\exists !$] S_0 initial state of ASM_C such that $conf(s_0, S_0)$;*

2. *for every change step (s, m, s') with s the current state of O_C, $\exists$ [$\exists !$] (S, S') step of ASM_C with S the current state of ASM_C, such that (s, m, s') is* step conforming *with (S, S');*

3. *no specification invariant of ASM_C is ever violated.*

Note that, in case of deterministic systems, Def. 11.13 (both the weak and the strong versions) coincides with Def. 11.12. So, we can adopt Def. 11.13 as the general notion of runtime conformance of our framework.

Our monitoring system can only deal with strong conformance. Therefore, in case of non-determinism, during the runtime monitoring our system chooses, among the next states of the ASM, the unique state that conforms to the Java state. Fig. 11.3 depicts this situation: given the Java state s' produced by the execution of the method m, only one of the next states of the ASM (S'_j) is state conformant with s'.

$$\begin{tikzcd}
O_C & & & ASM_C & & \\
s \arrow[rrr, dotted, "conf"] \arrow[d, "m"] & & & S \arrow[dll, "step"'] \arrow[d, "step"] \arrow[drr, "step"] & & \\
s' \arrow[rrr, bend right, "conf(s',S'_j)"'] & S'_1 & \dots & \mathbf{S'_j} & \dots & S'_n
\end{tikzcd}$$

Figure 11.3: Strong conformant step

If there is more than one next state conformant (weak conformance), instead, the system does not know which one to choose and rises an exception.

11.2 CoMA implementation

We here describe how CoMA works. We provide technical details on how the runtime monitor has been implemented by exploiting the mechanism of Java annotations to link observable Java elements to corresponding ASM entities (Section 11.2.1), and AspectJ to observe code execution and establish conformance relation (Section 11.2.2).

As a supporting example we introduce the Java class `Counter`, shown in Code 11.1. It implements a counter that can be initialised to any value, is incremented through the method `inc`, and is read through the pure method `getCounter`.

We also introduce the ASM specification *counterMax10*, shown in Code 11.2. It models a counter limited to 10 and initialised to the monitored value *initValue*; *counter* and *initValue* are both 0-ary functions. The invariant checks that the counter is always less or equal to 10; note that the transition rules always respect the invariant, but the value taken by *initValue* in the initial states could produce a violation.

```
import org.asmeta.monitoring.*;

@Asm(asmFile = "models/counterMax10.asm")
public class Counter {
    @FieldToFunction(func = "counter")
    public int counter;

    @StartMonitoring
    public Counter(@Param(func = "initValue") int x) {
        counter = x;
    }

    @RunStep
    public void inc() {
        counter ++;
    }

    @MethodToFunction(func = "counter")
    public int getCounter() {
        return counter;
    }
}
```

Code 11.1: Java class `Counter` annotated for monitoring

11.2.1 Linking the Java code and the ASM with Java Annotations

Java annotations [102] are meta-data tags that can be used to add some information to code elements as class declarations, field declarations, etc.

In addition to the standard ones, annotations can be defined by the user similarly as classes. For our purposes we have defined a set of annotations in order to link the Java code to its abstract specification. The retention policy, i.e., the way to signal how and when the annotation can be accessed, of all of our annotations is `RUNTIME` – annotations can be read by the compiler and by the monitor at runtime through *reflection*. In Java, thanks to reflection, an executing program can examine itself at runtime, and manipulate some of its internal properties. In this case reflection give us the ability of parsing a Java class that must be monitored and reading the annotations of its members.

In the following we review the annotations we currently use in our monitoring framework.

@Asm In order to link a Java class C with its corresponding ASM model ASM_C, the class must be annotated with the `@Asm` annotation having the path of the ASM model as string attribute. The class `Counter`, shown in Code 11.1, is linked to the ASM specification *counterMax10*, shown in Code 11.2.

@FieldToFunction, @MethodToFunction To establish the mapping defined by the function *linkOS* (see Formula 11.1), we annotate each observed field $f \in OF(C)$ by `@FieldToFunction`, and each observed method $m \in OM(C)$ by `@MethodToFunction`; both these annotations have a string attribute yielding the name of the corresponding ASM function. In the example, the Java field `counter` and the Java pure method `getCounter` are both linked to the ASM function *counter* (obviously this double linking is redundant, but it is only for demonstration purposes).

@FieldToLocation, @MethodToLocation Actually the approach also permits to link fields and pure methods to locations, rather than functions. Annotations `@FieldToLocation` and `@MethodToLocation` permit to identify a location using two attributes: a string attribute to specify the name of the corresponding function, and another attribute (an array of strings) to specify the values for the function arguments.

```
asm counterMax10
import StandardLibrary

signature:
    dynamic controlled counter: Integer
    dynamic monitored initValue: Integer

definitions:

    invariant inv_max10 over counter: counter <= 10

    main rule r_Main =
        if counter < 10 then
            counter := counter + 1
        endif

default init s0:
    function counter = initValue
```

Code 11.2: ASM model of a counter limited to 10

@Param To establish the mapping defined by the function *linkOI* (see Formula 11.2), we annotate each observed constructor/method formal parameter with @`Param`. This annotation has a string attribute for specifying the name of a monitored function of the ASM. At runtime, when the constructor/method is executed, the value of the actual parameter is used to set the linked monitored function.

@StartMonitoring In order to define the set of observed constructors $OC(C)$ (see Def. 11.6), i.e., defining the starting points of the monitoring, the user has to annotate a not empty subset of constructors through the annotation @`StartMonitoring`[4]. In the example there is just one constructor whose parameter is linked (through the annotation @`Param`) with the ASM monitored function *initValue* which fixes the initial value of the counter (see the specification in Code 11.2).

@RunStep All methods of $changingMethods(C)$ are annotated with the annotation @`RunStep`. In the example, the only observed method is `inc()`.
If the Java class has more than a changing method, the code is *externally* nondeterministic, that is the order in which the methods are called is not predictable. If we want to use a deterministic specification for monitoring *externally* nondeterministic programs, we must provide a mechanism to inform the formal specification about which method has been selected. For this purpose, the annotation @`RunStep` has two optional arguments that permit to signal to the ASM which changing method has been executed: *setFunction* specifies the name of a 0-ary monitored function of the ASM model, and *toValue* the value to whom it must be set. Code 11.3 shows the Java class `CounterDec` with two changing methods, `inc` and `dec` that permit to increment and decrement a counter; the corresponding ASM in Code 11.4 increments or decrements the function *counter* according to the value of the monitored function *action* (*INC* or *DEC*). In the Java code, the @`RunStep` annotations of the changing methods `inc` and `dec` specify that the monitored function *action* must be set, respectively, to *INC* and *DEC*.

Considerations on the use of annotations

Our use of the annotation mechanism requires a very limited code modification and differs from that usually exploited in other approaches for system monitoring. Usually annotations are used to enrich the code with extra formal specifications to obtain behavioural information about the target program [43, 108]. This leads to the lack of separation between the implementation of the system and its high-level requirements specification. In our approach, the few annotations

[4]We do not consider the default constructor. If the class does not have any constructor, the user has to specify an empty constructor and annotate it with @`StartMonitoring`.

```
import org.asmeta.monitoring.*;

@Asm(asmFile = "CounterDec.asm")
public class CounterDec {
    @FieldToFunction(func = "counter")
    public int counter;

    @RunStep(setFunction = "action", toValue = "INC")
    public void inc() {
        counter ++;
    }

    @RunStep(setFunction = "action", toValue = "DEC")
    public void dec() {
        counter −−;
    }
}
```

Code 11.3: Java – Counter with decrement

```
asm CounterDec
import StandardLibrary

signature:
    controlled counter: Integer
    monitored action: {INC | DEC}

definitions:

    main rule r_Main =
        if action = INC then
            counter := counter + 1
        else
            if action = DEC then
                counter := counter −1
            endif
        endif
```

Code 11.4: ASM – Counter with decrement

are only used to link the code to its specification, but keeping them separate. Furthermore, annotations are statically type checked and since the annotations are read reflectively at runtime, the monitoring setup can be carried out very easily. This is much more convenient than inserting special comments (like JML) and writing our own parser for them. Moreover, Java annotations make the links more robust when code refactoring is applied. Our approach fosters the reuse of specifications when code changes.

11.2.2 Implementation of the runtime monitor through AspectJ

The *runtime monitor* (see Fig. 11.1) is implemented through the facilities of AspectJ that permits to easily observe the execution of Java objects. In Section 11.2.2.1 we see the structure of the AspectJ *aspect* we use in our framework, and in Section 11.2.2.2 how it orchestrates the monitoring process.

11.2.2.1 Development of the aspect

AspectJ allows programmers to define special constructs called *aspect*s. By means of an aspect, AspectJ allows to specify different *pointcut*s, that are points of the program execution one wants to capture; for each pointcut it is possible to specify an *advice*, that is the actions that must be executed when the pointcut is reached during the execution of the program. AspectJ permits to specify when to execute the advice: *before* or *after* the execution of the code specified by the pointcut.

The CoMA tool supports two different ways, *built-in* and *compiled*, of obtaining the aspect to be used for monitoring.

Built-in In this approach there is just one aspect that permits to monitor all the objects of the classes that must be monitored:

(i) the pointcuts are general enough to capture the instantiations and the method executions of all the objects that must be monitored;

(ii) the advices are able to dynamically inspect the Java and the ASM state in order to do the conformance checking.

The main advantage of this approach is that the developer does not have to care about building the aspect. After having written the Java class and the ASM specification, she simply has to link them properly and add to the build path the general aspect we provide. Then she can execute the code immediately.

The main disadvantage of this approach is that, since the provided aspect is very general, it introduces an overhead in the pointcuts and in the advices that execute the conformance checking. For instance, the pointcuts to detect the creation of an observed object and to capture the execution of a changing method (we do not consider changing methods that are executed in the scope of other changing methods) are reported below.

```
pointcut objCreated(): call(@StartMonitoring *.new(..));
pointcut runStepCalled(): call(@RunStep * *.*(..)) && !cflowbelow(call(@RunStep * *.*(..)));
```

These pointcuts capture the calls of *any* method annotated with the specified annotations (i.e., @`StartMonitoring` and @`RunStep`).

In this approach, in order to obtain the observed state (see Def. 11.4), we use Java reflection: we read the values of the observed fields (those in $OF(C)$), and we execute the observed methods (those in $OM(C)$) and read the returned values.

Actually, in order to read the values of the observed fields, we have implemented (and experimented) two techniques[5]:

1) reading them through reflection at the beginning and at the end of the execution of a changing method;

2) using the AspectJ pointcut *set* in order to capture all their updates. In the following we report the pointcut we developed to capture the updates (and the values specified in the updates) of all the fields annotated with `@FieldToFunction` or `@FieldToLocation`.

   ```
   pointcut observedFieldSet(Object value, Object field):
           (set(@FieldToFunction * *) || set(@FieldToLocation * *)) &&
           args(value) && target(field);
   ```

The main advantage of using the first technique is that we can get their values only once for each changing method execution; using the second technique, instead, every time an observed field is updated we collect its value: if a field is updated frequently (e.g., in a loop), using the *set* pointcut the performances of the monitoring module can get worse. However, the *set* pointcut can read private fields without programmatically changing their visibility.

Compiled In this approach, for each Java class that must be monitored, a suitable aspect is built.

The main advantage of this approach is that the pointcuts definitions can be more precise. For example, the pointcut that captures the execution of the changing methods can specify exactly the methods whose execution must be captured. In the *built-in* approach, instead, we can only specify that the methods must be annotated with @`StartMonitoring` or @`RunStep`. For instance, the pointcuts for the class `Counter` are

```
pointcut objCreated(): call(Counter.new(int));
pointcut methodCalled(): call(public void Counter.inc());
pointcut runStepCalled(Counter target): methodCalled() && !cflowbelow(methodCalled()) &&
                                        target(target);
```

where we can exactly specify that we want to capture the creation of an object of the class `Counter` made through the constructor with an integer parameter (pointcut `objCreated`), and the calls to method `inc` (pointcut `runStepCalled`).
Also the advices definitions can be more precise. For example, to obtain the observed state, we do not need any more to reflectively access to the observed fields and methods, but we can directly read the values of the observed fields and invoke the observed methods using their identifiers.

The main disadvantage of the approach is that the developer, before running her code, must build the aspect: if the Java code and/or the ASM specification change, the aspect may need to be rebuilt.

[5]Note that the difference between the two techniques is only related to the way in which the observed fields are read: indeed, in both techniques, the returned values of the observed methods are always obtained through reflection.

11.2.2.2 Monitoring execution

Let's see how the aspect described in Section 11.2.2.1 permits to implement the monitoring framework depicted in Fig. 11.1.

Observer The pointcuts `objCreated` and `runStepCalled` permits to implement the *Observer*: when they detect that an observed constructor or a changing method has been called, a proper advice is activated.

Simulator When the Observer detects that an object that must be monitored has been created (pointcut `objCreated`), an instance of the AsmetaS simulator (see Section 4.1) is created for the corresponding ASM and, if any parameter is annotated with @`Param`, the corresponding monitored function is set with the value of the actual parameter.

Upon the execution of a changing method m signaled by the Observer (pointcut `runStepCalled`), the *Simulator* performs an ASM step by AsmetaS:

1. before the execution of m, an advice executes the following activities:
   (a) if the @`RunStep` annotation of m defines the arguments *setFunction* and *toValue*, it sets, in the ASM simulation, the monitored function specified by *setFunction* to the value specified by *toValue*; moreover, for any parameter annotated with @`Param`, the corresponding monitored function is set with the value of the actual parameter;
   (b) it requests the *Analyser* to do a state conformance check ($conf(s, S)$ in Def. 11.11);
2. after the execution of m, another advice executes the following activities:
   - if the ASM is deterministic:
     (a) it simulates a step of the ASM;
     (b) it asks the Analyser for checking again the state conformance ($conf(s', S')$ in Def. 11.11).
   - if the ASM is nondeterministic:
     (a) it asks the Simulator for all the next states $nextStates(s)$ of the ASM[6];
     (b) it asks the Analyser for checking if it exists a unique $s' \in nextStates(s)$ such that state conformance holds ($conf(s', S')$ in Def. 11.11);
     (c) if a single s' is found, it moves the ASM under simulation to s'.

Analyser The *Analyser* compares the Java and the ASM states. To check state conformance (see Def. 11.10), we have implemented the conformance relation $\overset{conf}{=}$ among Java and ASM values as a string comparison. Therefore, the Java and the ASM values are both transformed into strings for comparison. If the conformance does not hold it raises an exception of non-conformity. For example, an object of the Java class `Counter` (see Code 11.1) could result non-conformant since the field `counter` can assume a value greater than 10, whereas in the corresponding ASM specification (see Code 11.2) the value of the linked function *counter* is bounded to 10.

As seen previously, in case of nondeterministic systems, the Analyser also checks if, among the next states of the ASM, there is one and only one state that is state conforming with the Java state. If it does not find any state, it raises an exception of non-conformity. If more that a state is found, it raises another kind of exception, indicating that it can not ensure strong conformance.

[6]In order to obtain all the next states of a state during simulation, we had to extend the AsmetaS simulator. The only source of nondeterminism, that can result in having more than a possible next state, is the presence of at least a choose rule in the model. The number of next states is determined by the *degree* of nondeterminism of the choose rules of the model (i.e., the number of values for which the choose rules guards are satisfied). Since the normal version of the simulator already requires that the domains of the logical variables in the choose rules are finite, we are sure that the number of next states is finite (although it may be big).
A similar extension to an ASM simulator has been made in [23], where the CoreASM simulator is used to do model checking of ASMs.

	Java	JML	JavaMOP	CoMA			
				AsmetaS	aspect		
					built-in - reflection	built-in - set	compiled
Counter	4	280	(FSM) 109	4837	+ 898	+ 825	+ 783
Iterator	8	N/A	(FSM) 91	866306	+ 1820	+ 1812	+ 1439
Init. Order Fiasco	7	N/A	(LTL) 72	870719	+ 2366	+ 2235	+ 1914

Table 11.1: Execution time in the experiments (in secs)

11.3 Evaluation

In order to assess the viability of our approach, we have taken several examples in literature and checked whether we were able to apply our approach to existing runtime case studies, including the Railroad Gate [53], the Static Initialisation Order Fiasco problem [22], a robotic assembly system [126], the Knight's Tour problem [166]. We have written the Java code, if not available, and their ASM specifications (see [8] for details). We applied also CoMA to several Java programs borrowed from JavaMOP (see Section 10.2.2), like `Iterator` and `FileWriter`. Overall we found our approach applicable to all the considered case studies.

11.3.0.3 Execution time

In order to evaluate the runtime overhead of our approach, we have considered three examples, the *Counter*, the *Iterator* and the *Initialisation Order Fiasco*[7], and we have monitored them with CoMA, JavaMOP (using FSM or LTL specifications), and JML (when applicable). A comparison with the approach we have presented in Section 10.2.3 is not possible. They use, like CoMA, interpretation of formal specifications, but their tool is not available and no time data are published.

All the Java programs are correct and accessed correctly, i.e., if monitored, they conform to their formal specifications. Indeed, our aim here is to measure the overhead introduced by our runtime framework in a normal program execution.

Table 11.1 reports the average time over 20 runs of the experiment in which 100 instances of the class under monitoring run in parallel for 1000 steps. JML can not be used with the *Iterator* and the *Init. Order Fiasco* since it is not able to express the required properties. Column *Java* reports the time taken by the code under analysis.

For the CoMA, Table 11.1 reports the overall time divided between the time taken by the simulator (column *AsmetaS*), and the time taken by the monitor (column *aspect*). For the time taken by the monitor, we report the times of the three kinds of aspects described in Section 11.2.2.1: *built-in* (using *reflection* or the *set* pointcut for reading the observed fields) and *compiled*.

It is apparent that most of the time is taken by the simulator, which has never been optimized for performances and, instead, uses technologies that can be time consuming (e.g., it has been built on the top of the Eclipse Modelling Framework, and it widely uses design patterns for visiting the ASM under simulation). A solution for this problem could be to translate the ASM machine directly into Java code (similarly of what is done in JavaMOP [45] and in LIME [108]). However, encoding ASMs into Java would require the proof of the semantic correctness of the translation.

We discovered that, if we use the *built-in* aspect, using *reflection* or the pointcut *set* for reading the observed fields is almost equivalent. However, in another experiment we discovered

[7]We have described the *Static Initialisation Order Fiasco* problem in Section 10.2.1. It is a problem that arises in C++ programs, and it is caused by the fact the initialisation of static objects is executed in a nondeterministic order. In our experiments we have provided a Java simulation of the setting in which the problem can arise.

```
CheckCounter(Counter c) {
    int count = 0;// counter value

    // inc call event
    event inc before(Counter c): call(∗ Counter.inc()) && target(c)
                                  {count ++;}
    // error event
    event err after(Counter c): call(∗ Counter.inc()) && target(c) &&
                                 condition(c.getCounter() != count) {}
    // the FSM
    fsm: safe [inc −> safe    err −> error]        error []

    @error {
        System.out.println("Counter not incremented");
    }
}
```

Code 11.5: JavaMOP – FSM specification for the Java class `Counter`

that the pointcut *set* performs worst when an observed field is updated frequently (e.g., in a loop). Instead, as expected, compiled aspects provide the best results since they are tailored on the specific classes under monitoring.

Compared to JML and JavaMOP our approach is always more slow. However, although our approach is not competitive with others in terms of time overhead, we believe that it provides several advantages (explained in the next sections) and it can be used when performances are not critical.

11.3.0.4 Usability and expressiveness

Although any comparison of our approach with others in terms of usability and expressiveness may be disputable, since it may depend on the expertise and taste of the user, some general considerations follow.

In comparison with JML, CoMA can be used to express the behaviour of a single method call and also the interaction among calls, while JML concentrates on single methods. There exist, however, JML extensions that allow the specification of temporal aspects of Java interfaces (like LIME [108] and trace assertions of Jass [35]). Another difference is that CoMA has a model separated from the implementation, while JML follows a unique model paradigm in which the code itself contains its specification. The advantage of CoMA is that the specification can exist even before its implementation and can be used for several preliminary activities (like model simulation, model review, and formal verification).

The expressiveness of CoMA is greater than approaches using plain FSMs, since ASMs can have infinite states and can be viewed as pseudo-code over abstract data type. In many approaches, like in JavaMOP and in JavaMAC (which uses automata with auxiliary variables) [123], FSMs need to be enriched with state variables. For instance, the FSM specification for the class `Counter` that must be used in JavaMOP is shown in Code 11.5. In order to check if the counter is incremented correctly, a variable `count` must be used to record the expected value of the counter (as we did in the ASM model 11.2 using the function *counter*).

Since JavaMOP specifications are compiled into AspectJ[8], JavaMOP can include and use all the power of AspectJ (e.g., they can define events as AspectJ pointcuts). However, we believe that mixing implementation and specification notations may encourage the user to insert implementation details in the specification at the expense of abstractness. An important feature of our methodology is the clear separation between the monitored implementation and the high level specification also in terms of notation, as in [123, 126].

[8]Note that we use AspectJ only to drive the monitoring, not to encode the specification.

11.3.0.5 Comparison with approaches using declarative notations

In this work, we assume that the specification is given in operational style instead of the more classical declarative style.

An objective comparison with approaches that specify correctness properties using declarative notations is questionable. There has been an endless debate about which style fits better the designer needs: some argue that with an operational style the designers tend to insert implementation details in the abstract specifications, others observe that practitioners feel uncomfortable with declarative notations like temporal logics.

The scope of our work is to provide evidence that also abstract operational notations can be effectively used for runtime monitoring. Sometimes, operational specifications are easier to write and understand; other times, declarative specifications are preferable. For instance, LTL and PTLTL (Past Time LTL) can describe correct sequences of method calls with ease. As seen in Section 10.2.2 for JavaMOP, the correct order of calls for an `Iterator` (the method `next` can be called only if, previously, the method `hasNext` has been called and it has returned *true*), is specified by the following PTLTL formula:

$$\Box(\mathit{next} \implies \odot\, \mathit{hasNextTrue})$$

where the operator $\odot$ means "in the previous time step", the atomic predicate *next* signals if the method next has been called, and *hasNextTrue* signals if the method `hasNext` has been called and it has returned *true*. In this case, the property is very concise: expressing it with an operational specification would be less concise and it would require to add supporting variables to memorize which methods have been called. However, properties about states are more difficult (and sometimes impossible) to write. For instance, the fact that an unbounded counter is correctly incremented is not expressible by LTL. Indeed, LTL does not allow variable quantifiers and, therefore, formulae like

$$\forall x\, \Box(\mathit{counter} = x \implies \bigcirc(\mathit{counter} = x + 1))$$

are incorrect. In ASMs, instead, such property is expressible very easily.

11.4 Monitoring web services through CoMA

In order to test our approach in a real application setting, we applied it to the web services scenario.

Web services can be seen as particular *component-based systems. Component-based software engineering* (CBSE) is a reuse-based approach to software systems development [157] whose aim is to produce *independent components* that are completely specified by their interfaces. The implementation of a component should be separated from its interface so allowing to substitute a component without affecting the overall system. A problem that arises in CBSE is how to assure that components behave as expected. As stated in [157], *a viable solution is to certify that components conform to a formal specification.* We want to show here that runtime monitoring (through CoMA) can be a solution for certifying the conformance. Our approach permits to check, not only the correctness of the implementation, but also that the component is accessed in the right way (e.g., interface contracts on the calling order of the interface services are respected).

We have developed an e-commerce web service using Apache Axis2 [148, 4], a framework for the development of web applications in Java. The web service, shown in Code 11.6, exposes two operations[9], *createCart* to create a cart, and *addItem* to add an item to the cart.

A correct usage of the web service requires that:

PROP1: the operation *addItem* is called only if the operation *createCart* has already been called, i.e., one item can be added to the cart only if the cart exists;

[9]In web services the provided interfaces are called *operations*. So, in the following, we refer to the changing methods as operations.

```
package org.ecommerce;
import org.asmeta.monitoring.*;

@Asm(asmFile="models/eshop.asm")
public class Eshop {
    private static int counter = 0;
    private String clientID;
    private int numberOfElements;

    @StartMonitoring
    public Eshop() {
        clientID = "client_" + counter;
        counter++;
        numberOfElements = 0;
    }

    @RunStep(setFunction = "operationCalled", toValue = "CC")
    public String createCart() {
        //creation of the cart (not reported here)
        return clientID + " − cart created";
    }

    @RunStep(setFunction = "operationCalled", toValue = "ATC")
    public String addItem() {
        numOfEls++;
        return clientID + " − item added − " + "# elements = " + numberOfElements;
    }

    @MethodToFunction(func = "elementsInCart")
    public int getNumberOfElements() {
        return numberOfElements;
    }
}
```

Code 11.6: E-shop web service

PROP2: no more than 5 items are added to the cart.

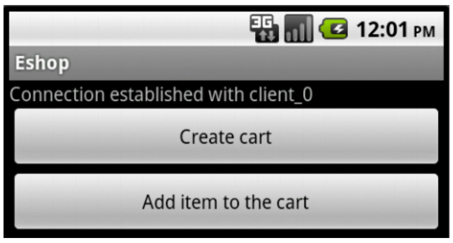
The assurance of both correctness properties is not guaranteed by the web service implementation. However their violations can be discovered if the web service is runtime monitored using, as formal specification, the ASM shown in Code 11.7.

In the ASM model, the operation chosen by the user is modeled through the monitored function *operationCalled* that can take value *CC* if the user wants to create a cart, and *ATC* if she wants to add one item to the cart. The controlled part of the ASM state is composed of the boolean function *cartCreated*, that records if a cart has been created, and the integer function *elementsInCart* that records how many items have been added to the cart. An invariant checks that items are added to the cart only if the cart has already been created.

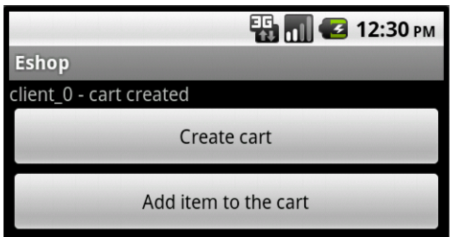
In order to test the web service, we have developed a client application for the Android platform [3]: the application is composed of two buttons that invoke the operations `addItem` and `createCart`, and a text box where the result of an operation execution is shown. As suggested in [89], the developed client is *faulty* since its usage can lead to the violation of the two correctness properties required by the web service. Indeed, we do not hide a button when it should not be called: in such way it is possible that the conversation with the web service is executed wrongly (e.g., an item is added to the cart before the cart has been created). Fig. 11.4a shows the client application when a connection with the web service has been established. Fig. 11.4b and 11.4c show the application when, respectively, the creation of the cart has been executed and an item has been added to the cart. In both cases the web service is invoked correctly and no correctness property is violated.

```
asm eshop
import StandardLibrary
signature:
    enum domain OperationCalledDomain = {CC | ATC}
    monitored operationCalled: OperationCalledDomain
    controlled cartCreated: Boolean
    controlled elementsInCart: Integer
definitions:
    rule r_createCart =
        if (operationCalled = CC) then
            cartCreated := true
        endif

    rule r_addToCart =
        if (operationCalled = ATC and elementsInCart < 5) then
            elementsInCart := elementsInCart + 1
        endif

    invariant inv_calledOperationsOrder over operationCalled:
                        operationCalled = ATC implies cartCreated

    main rule r_Main =
        par
            r_createCart []
            r_addToCart[]
        endpar

default init s0:
    function cartCreated = false
    function elementsInCart = 0
```

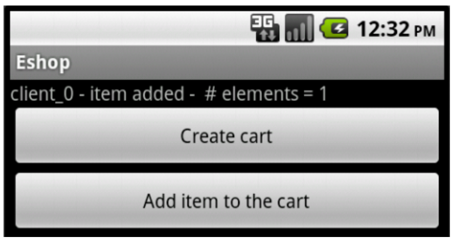
Code 11.7: ASM model of the e-shop web service


12:01 PM
Eshop
Connection established with client_0

(a) Client application loaded


12:30 PM
Eshop
client_0 - cart created

(b) Correct creation of the cart


12:32 PM
Eshop
client_0 - item added - # elements = 1

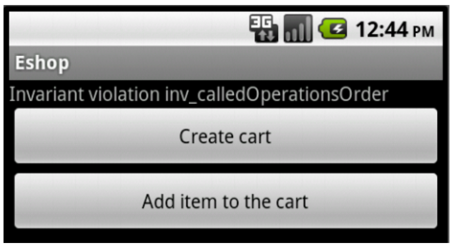
(c) Correct insertion of an item to the cart

Figure 11.4: Correct usage of the web service

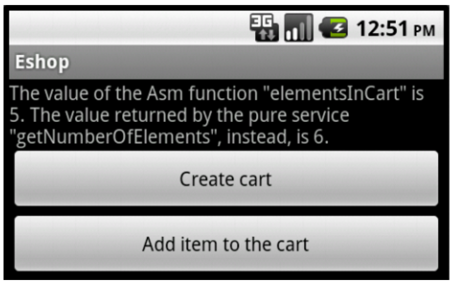
Let's see now how CoMA reveals the violation of **PROP1** and **PROP2**.

Violation of PROP1 The ASM has a boolean function *cartCreated* that records if a cart has been created, that is if the operation *createCart* has been called. The invariant contained in the machine checks that, when an item is added to the cart, the cart has already been created. Fig. 11.5a shows the error message that is shown when the "Add Item to the cart" button is selected firstly: the message is generated by the invariant violation during the ASM simulation.

Violation of PROP2 The violation of **PROP2** can be discovered when the Analyser of the runtime monitor checks for the state conformance after the execution of an operation. Fig. 11.5b shows the error message that is obtained if, after the correct creation of the cart, the "Add Item to the cart" button is selected 6 times. The web service does not bound the value of *numberOfElements*, whereas in the ASM the function *elementsInCart* is not incremented if it is greater than or equal to 5.


Invariant violation inv_calledOperationsOrder

(a) Violation of **PROP1** – Invariant exception


The value of the Asm function "elementsInCart" is 5. The value returned by the pure service "getNumberOfElements", instead, is 6.

(b) Violation of **PROP2** – Conformance exception

Figure 11.5: Wrong usage of the web service

Chapter 12

Using runtime monitoring for testing nondeterministic programs

In the software system life cycle, models are used to represent the system behaviour in a high-level abstract way. In case of underspecification – for instance because some implementation choices are left abstract – or not fully predictable systems, models result *internally* nondeterministic, i.e., given the same input sequences at different times, different output sequences can be produced. This distinguishes from *external* nondeterminism that is due to the unknown behaviour of the environment. The presence of internal nondeterminism makes all the common validation and verification activities more complex, although they still need to be performed. From now on we consider *nondeterminism* as *internal nondeterminism.*

Model-based testing (MBT) is accepted as a fully automated, flexible, and efficient technique to generate test cases that can lead to more effective testing [96]. MBT overcomes some limitations of the white box software testing. It addresses the *test oracle problem*, which is still an open problem in the context of software testing: in MBT, specifications are used as oracles since expected outputs are generated together with the inputs.

Research has resulted in numerous approaches differing in how test cases are generated from models. While the presence of nondeterminism is not a problem for some techniques (e.g., labeled transition systems [162]), it is a challenge for those approaches where test cases are linear sequences of execution states (e.g., approaches that derive test sequences from counterexamples returned by model checkers).

We want now to consider these latter approaches. Adapting the definitions in [73], we can define a test case $t = \langle s_1, s_2, \ldots, s_n \rangle$ as a path in the specification *spec* where each s_i (with $i = 2, \ldots, n$) is a next state of s_{i-1} and s_1 an initial state. Let $T(\mathit{spec})$ be the set of all the paths of *spec*. Given a path t, $\mathit{In}(t) = \langle \mathit{In}(s_1), \mathit{In}(s_2), \ldots, \mathit{In}(s_n) \rangle$ represents the input sequence (i.e., the moves of the environment on the specification) and $\mathit{Out}(t) = \langle \mathit{Out}(s_1), \mathit{Out}(s_2), \ldots, \mathit{Out}(s_n) \rangle$ the output sequence (i.e., the reaction of the specification to the application of the inputs). Let's call $\mathit{Act}(t) = \langle \mathit{Act}(s_1), \mathit{Act}(s_2), \ldots, \mathit{Act}(s_n) \rangle$ the output sequence produced by the implementation when executed using $\mathit{In}(t)$ as inputs.

The implementation passes a test t ($\mathit{pass}(t)$) if, executed using inputs $\mathit{In}(t)$, it returns the expected outputs, i.e.,

$$\mathit{pass}(t) \iff \mathit{Act}(t) = \mathit{Out}(t) \tag{12.1}$$

The definition of failure for deterministic systems derives straightforwardly from Formula 12.1. An implementation fails a test t ($\mathit{fail}(t)$) if, executed using inputs $\mathit{In}(t)$, it does not return the expected outputs, i.e.,

$$\mathit{fail}(t) \iff \mathit{Act}(t) \neq \mathit{Out}(t) \tag{12.2}$$

So, for deterministic systems, if the implementation does not pass a test, it means that it fails it. However, if the specification is nondeterministic, definition of $\mathit{fail}(t)$ as described in Formula

12.2 is no more valid. In fact, the implementation could deviate from test case t, taking a different but valid execution path t'. In this case, we say that the test case *falsely* fails (*falselyFail*(t)), i.e.,

$$\mathit{falselyFail}(t) \iff \big(\mathit{Act}(t) \neq \mathit{Out}(t) \wedge \exists t' \in T(\mathit{spec})\,\big[\mathit{Act}(t) = \mathit{Out}(t')\big]\big) \tag{12.3}$$

This means that the implementation has executed a path that is a valid path in the specification, but that is different from the path selected as test case. These tests are usually called *inconclusive*. So, in order to deal with nondeterministic specifications, definition of *fail* as given in Formula 12.2 must be refined in the following way:

$$\mathit{fail}(t) \iff \big(\mathit{Act}(t) \neq \mathit{Out}(t) \wedge \neg\exists t' \in T(\mathit{spec})\,\big[t' \neq t \wedge \mathit{Act}(t) = \mathit{Out}(t')\big]\big) \tag{12.4}$$

During the test execution, at each step i, we observe if the actual output $\mathit{Act}(s_i)$ is as the expected output $\mathit{Out}(s_i)$: if not, we say that there is a *deviation*. In case of deviation, since we do not know if the implementation actually failed or if the test was inconclusive, we must stop the test case execution without giving any response.

A technique that tries to address the problem of inconclusive tests is presented in [74, 73], in which the test case generation process using model checkers is extended in order to deal with nondeterminism. The authors present a technique that identifies what are the nondeterministic choices taken in a counterexample: such technique permits to discover if a deviation exists from the expected output during a test case execution due to a nondeterministic choice (inconclusive test). Starting from an inconclusive test, the proposed process can iteratively build a *tree-like* test case in which the alternative valid branches of a computation are considered. They also extend common coverage criteria for deterministic systems to nondeterministic systems.

The approach we here present tries to address the problem of inconclusive tests in a different way from that presented in [74, 73]. We combine model-based testing, used here to automatically generate, from nondeterministic specifications, only the inputs of the test cases (*test data*), with *runtime monitoring* (see Chapter 10) which is used to provide an oracle (the expected outputs) that never bears inconclusive responses. So, the idea of our approach is to ignore the expected outputs of a test sequence (since they could lead to inconclusive responses), and providing the test oracle through a runtime monitoring technique that is able to assess conformance between an implementation and a nondeterministic specification.

Among the different techniques existing for MBT, we here consider the technique that uses the model checkers capability to generate a counterexample upon a *trap property* violation [75, 71], and that interprets counterexamples as tests. Nondeterminism is not a problem for a model checker itself, but this technique, since the counterexamples are sequences of states, suffers from the problem of inconclusive tests in case of nondeterministic specifications. This is solved by monitoring at runtime the execution of a test case by checking, at each step, the conformance of the code w.r.t. its specification, even at the nondeterministic points. This permits us to avoid stopping the test execution and discarding the test, when the test deviates from the expected outputs. A further advantage is that we are able to check, at each step, which testing requirements are achieved, so having a measure of adequacy and avoiding redundant tests.

On the other hand, also runtime monitoring can benefit from our approach. Runtime monitoring does not suffer from the test oracle issue, but, if one wants to use it for testing, there is still the problem of selecting relevant inputs, and of measuring the confidence that the runtime monitoring covered all the possible system behaviours.

We use ASMs (see Section 2.1) as formal method for specification purposes and Java as implementation language. In [77] a model-based testing technique for deterministic ASM models has been presented, while the runtime monitoring of Java programs w.r.t. corresponding ASM specifications has been described in Chapter 11.

We select the Tic-Tac-Toe game as example of a system with nondeterministic behaviour (both at specification and code levels). To assess the quality of our approach and to measure its fault detection capability, we apply mutation analysis.

The chapter is organized as follows: Section 12.1 introduces the running case study, its ASM formal specification, a Java implementation, and their link using CoMA. Section 12.2 presents the necessary background on model-based testing using model checker for deterministic ASMs [76, 77] and its extension to deal with nondeterministic ASMs. Section 12.3 explains our approach to combine model-based testing with runtime monitoring, while Section 12.4 reports our experiments on the Tic-Tac-Toe example.

12.1 Running case study – A simple nondeterministic model

As motivating example and running case study, we consider a Tic-Tac-Toe game where a human player challenges a computer program. The requirements include that only valid moves are accepted, i.e., each player can put her symbol (nought or cross) only in an empty cell and when it is her turn, until one wins. The system must be able to identify valid moves and ignore invalid moves, check if one player wins and if the game is tie. The user moves are monitored by the system, while the program decides its moves according to some strategies. At specification level, the designer does not want to detail how the computer will play, since the strategy may be complex, change in order to improve performances, and include some random choices. The computer decisions will be left unspecified as nondeterministic choices. However, the designer wants to be sure that the implementation satisfies the requirements of correctness listed above.

12.1.1 ASM formal specification

Code 12.1 shows the ASM specification of the Tic-Tac-Toe. The binary function *board* models the board of the game: every location represents a cell of the board, and it can *contain* a sign (*CROSS* or *NOUGHT*) or *be* empty (*EMPTY*). Function *result* records if there is a winner (*U_WON* or *C_WON*), if the game is tie (*TIE*), or if the game is still running (*PLAYING*). The monitored function *action* models the intention of a player to make a move. The move of the player *user* is modeled through macro rule *r_moveUser* in which the coordinates of the chosen cell are given to the machine through the monitored functions *uSelRow* and *uSelCol*. The move of the player *computer*, instead, is done in the macro rule *r_moveComp* in which an empty cell is nondeterministically chosen (no particular strategy is adopted).

Note that the ASM model correctly describes the requirements described above. In fact only valid moves are accepted: a) if the wrong player wants to move (value of the monitored function *action*), the request is ignored, b) if the player *user* chooses a not empty cell, the move is refused. The identification of the winner and of the player that must play, and the moves of the player *computer* are correct as well. Finally, as requested, the *computer* decisions are left totally nondeterministic, so that any possible strategy for the player *computer* in the implementation can be captured by the abstract specification.

12.1.2 Java implementation

Code 12.2 shows a possible Java implementation (unnecessary details have been omitted). The board is represented with the two-dimensional array `board` whose type is the enumerative `Sign` = {`UNDEF`, `CROSS`, `NOUGHT`}. Method `execUserMove` permits to execute a user move, by specifying the selected cell in the method parameters. Method `execUserMove` executes a computer move, by randomly choosing an empty cell. Note that here any strategy could have been implemented: we choose this trivial strategy for the sake of conciseness.

It is easy to see that the Java code satisfies the requirements: no invalid moves are executed, and a player can move only if it is its turn.

12.1.3 Runtime monitoring through CoMA

In order to be runtime monitored with CoMA (see Chapter 11), the Java implementation (Code 12.2) has been linked to its formal specification (Code 12.1) using Java annotations, as described

```
asm ticTacToe
signature:
    domain Coord subsetof Integer
    enum domain Sign = {CROSS | NOUGHT | EMPTY}
    enum domain Status = {TURN_USER | TURN_COMP}
    enum domain ActionDomain = {U_MOVE | C_MOVE}
    enum domain ResDom = {PLAYING | U_WON | C_WON | TIE}
    //first argument is the row, second argument is the column
    controlled board: Prod(Coord, Coord) −> Sign
    controlled status: Status
    monitored uSelCol: Coord
    monitored uSelRow: Coord
    monitored action: ActionDomain
    controlled result: ResDom
    controlled numOfMoves: Integer
    derived winOnRow: Prod(Coord, Coord, Sign) −> Boolean
    derived winOnCol: Prod(Coord, Coord, Sign) −> Boolean
    derived winOnDiag: Prod(Coord, Coord, Sign) −> Boolean
definitions:
    domain Coord = {0..2}
    //derived functions definition

    rule r_makeMove($r in Coord, $c in Coord, $s in Sign) =
        par
            board($r, $c) := $s
            numOfMoves := numOfMoves + 1
            if(winOnRow($r, $c, $s) or winOnCol($r, $c, $s) or winOnDiag($r, $c, $s)) then
                if($s = CROSS) then
                    result := U_WON
                else if($s = NOUGHT) then
                    result := C_WON
                endif endif
            else if (numOfMoves = 8) then
                result := TIE
            endif endif
        endpar

    rule r_moveUser =
        if (status = TURN_USER and board(uSelRow, uSelCol) = EMPTY) then
            par
                r_makeMove[uSelRow, uSelCol, CROSS]
                status := TURN_COMP
            endpar
        endif

    rule r_moveComp =
        if(status = TURN_COMP) then
            par
                choose $r in Coord, $c in Coord with board($r, $c) = EMPTY do
                    r_makeMove[$r, $c, NOUGHT]
                status := TURN_USER
            endpar
        endif

    main rule r_Main =
        if(result = PLAYING) then
            if(action = U_MOVE) then
                r_moveUser[]
            else
                r_moveComp[]
            endif
        endif

default init s0:
    function status = TURN_USER
    function board($r in Coord, $c in Coord) = EMPTY
    function result = PLAYING
    function numOfMoves = 0
```

Code 12.1: ASM specification of Tic-Tac-Toe

in Section 11.2.1.

The observed state is composed of the fields `board`, `movesExecuted`, `status`, and the pure

```
import org.asmeta.monitoring.*;

@Asm(asmFile = "models/ticTacToe.asm")
public class TicTacToe {
    @FieldToFunction(func = "board")
    public Sign[][] board;
    private Random rnd;
    @FieldToFunction(func = "numOfMoves")
    public int movesExecuted = 0;
    @FieldToFunction(func = "status")
    public Status status;
    private Sign winner;

    @StartMonitoring
    public TicTacToe() {...}

    @RunStep(setFunction = "action", toValue = "U_MOVE")
    public void execUserMove(@Param(func="uSelRow") int r, @Param(func="uSelCol") int c) {
        if(winner == null && status == Status.TURN_USER && board[r][c] == Sign.UNDEF) {
            board[r][c] = Sign.CROSS;
            movesExecuted++;
            status = Status.TURN_COMP;
            if(checkWinner(r, c, Sign.CROSS))
                winner = Sign.CROSS;
        }
    }

    @RunStep(setFunction="action", toValue = "C_MOVE")
    public void execComputerMove() {
        if(winner == null && movesExecuted < 9 && status == Status.TURN_COMP) {
            int r, c;
            do {
                r = rnd.nextInt(3);
                c = rnd.nextInt(3);
            } while(board[r][c] != Sign.UNDEF);
            board[r][c] = Sign.NOUGHT;
            movesExecuted++;
            status = Status.TURN_USER;
            if(checkWinner(r, c, Sign.NOUGHT))
                winner = Sign.NOUGHT;
        }
    }

    @MethodToFunction(func = "result")
    public String getWinner() {
        //depending on the values of "winner" and "numOfMoves", it returns "USER_WON",
        //"COMPUTER_WON", "TIE" or "PLAYING"
    }

    private boolean checkWinner(int r, int c, Sign sign) { ... }
}
```

Code 12.2: Java code of Tic-Tac-Toe

method `getWinner`, that are linked to the ASM functions *board*, *numOfMoves*, *status*, and *result*. The changing methods are `execUserMove` and `execComputerMove`; the ASM is aware of what method has been executed thanks to the monitored function *action* linked in the annotations of the two changing methods.

After having run the implementation with CoMA several times, we are confident that it is conformant to the formal specification.

Note that, since both the Java code and the ASM are nondeterministic, the notion of runtime conformance used is that given in Def. 11.13. The code is monitorable because each nondeterministic choice is strongly conformant, i.e., it exists only one next state of the specification that is conformant. Nondeterministic steps are caused by the execution of method `execComputerMove`. Fig. 12.1 shows an example of Java execution in which the computer, given a board configuration, chooses to place a nought in (2, 0): among the possible next states of the ASM, only one is conformant.

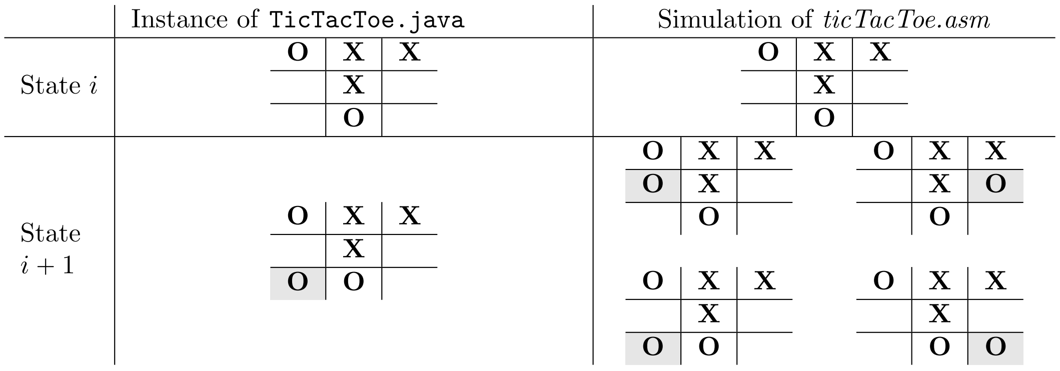


Figure 12.1: Runtime monitoring of `TicTacToe` through CoMA – Strong conformant step

12.2 Model-based testing for ASMs

In the context of the ASMs, one of the main applications of MBT consists in automatically generating tests from ASM models [76].

In the following we give some basic definitions about test generation from ASMs.

Definition 12.1 (Test sequence)**.** *A* test sequence *(or* test*) is a finite sequence of states $s_1, \ldots, s_n$ whose first element s_1 is an initial state, and each state s_i (with $i \neq 1$) follows the previous one s_{i-1} by applying the transition rules. The final state s_n is the state where a test goal is achieved.*

Definition 12.2 (Test suite)**.** *A* test suite *(or* test set*) is a finite set of test sequences.*

Definition 12.3 (Test predicate)**.** *A* test predicate *is a formula over the state and determines if a particular testing goal is reached. A coverage criterion C is a function that, given a formal specification, produces a set of test predicates. A test suite TS satisfies a coverage criterion C if each test predicate generated with C is satisfied in at least one state of a test sequence of TS.*

Definition 12.4 (Rule guard)**.** *Given a rule r_i, its* guard *g_i is the conjunction of the guards of the rules (conditional, forall and choose rules) that leads to the execution of r_i.*

For example, in the ASM model shown in Code 12.1, the guard of the macro call rule *r_moveUser[]* in the main rule is *result = PLAYING and action = U_MOVE.*

In [76] the *choose* rule is not considered. So we have to extend that work for dealing with choose rules. In particular, we must define how a choose rule contributes to the building of the guards of the rules that are in its scope. Let's define a general choose rule as

```
choose $x_1 in D_1, ..., $x_n in D_n with cond($x_1, ..., $x_n) do
    R[$x_1, ..., $x_n]
```

The guard of any rule contained in the scope of R (R included) must contain, as contribution of the choose rule, the following logic expression:

$$\bigvee_{(d\_1,\ldots,d\_n)\in(D\_1\times\ldots\times D\_n)} cond(d\_1, \ldots, d\_n) \tag{12.5}$$

Formula 12.5 simply requires that it exits a tuple of values for the variables of the choose rule such that its guard is satisfied (and so rule R can fire).

12.2.1 Coverage criteria for ASMs

Several coverage criteria have been defined for ASMs [76]. Some of them are more tailored on ASMs (Section 12.2.1.1), while others are more general and have been derived by those used in software coverage (Section 12.2.1.2).

```
ifT_ifF_ifF : result = PLAYING and action != U_MOVE and status != TURN_COMP
```

Figure 12.2: Test predicate for covering at false the guard of the conditional rule in *r_moveComp*

12.2.1.1 Criteria tailored on ASMs

One of the basic criteria for ASMs is the *rule coverage.* A test suite satisfies the *rule coverage* criterion if, for every rule r_i of the ASM, there exists at least one state in a test sequence in which r_i fires and there exists at least a state in a test sequence in which r_i does not fire. So, for each rule r_i, two test predicates are produced, g_i and $\neg g_i$.

Other coverage criteria tailored on ASMs require that, for example, every update is executed non-trivially (*rule update coverage*), or that every combination of n rules can be fired (*n-parallel rule coverage*). The *2-parallel rule coverage* criterion is useful for discovering inconsistent updates.

12.2.1.2 General criteria

Some other coverage criteria, instead, are more general and have been derived from those used in software coverage, as *decision coverage*, *condition coverage* and *modified decision condition coverage* (MCDC).

For example, a test suite satisfies the *decision coverage* criterion if, for every decision d_i of a rule r_i (e.g., the guard of a conditional rule), there exists at least one state in a test sequence in which r_i fires and d_i evaluates to *true*, and there exists at least a state in a test sequence in which r_i fires and d_i evaluates to *false*. So, for each decision d_i, two test predicates are produced, $g_i \wedge d_i$ and $g_i \wedge \neg d_i$.

For instance, the test predicate for the coverage at *false* of the guard of the conditional rule in *r_moveComp* (see Code 12.1), is the predicate shown in Fig. 12.2. The test predicate requires that the conditional rule of *r_moveComp* is executed: in the main rule, the guard of the outer conditional rule must be true and the guard of the inner conditional must be false. Moreover, since we are covering at *false*, the guard of the conditional rule in *r_moveComp* must be false.

12.2.2 Test generation by model checking

In order to build test suites satisfying some coverage criteria, we use a technique based on the capability of the model checkers to produce counterexamples. The method consists of the following steps:

1. The test predicates set $\{tp_i\}$ is derived from the ASM according to some desired coverage criteria.

2. The ASM specification is translated into the language of the model checker.

3. For each test predicate tp_i, the *trap property* `never`(tp_i) is proved. If the model checker finds a state s where tp_i is true, it stops and returns as counterexample a state sequence leading to s: from such sequence it is possible to build a test for covering tp_i. If the model checker explores the whole state space without finding any violation of the trap property, then the test predicate is said *unfeasible* and it is ignored. In the worst case, the model checker terminates without exploring the whole state space and without finding a violation of the trap property (i.e., without producing any counterexample), usually because of the state space explosion problem. In this case of model checker *empty* result, the user does not know if the test predicate is unfeasible or if a test exists but it is too difficult to find.

In order to derive test sequences from ASM models, we here use the ATGT tool [77], based on the model checker SPIN [99]. For this work, the tool has been extended in order to translate also ASM models containing choose rules.

```
--------------          --------------          --------------
State 1                 State 2                 State 3
--------------          --------------          --------------
result  = PLAYING       result  = PLAYING       result  = PLAYING
status = TURN_USER      status = TURN_COMP      status = TURN_USER
action = U_MOVE         action = C_MOVE         action = C_MOVE
uSelRow = 1             uSelRow = 2             uSelRow = 2
uSelCol = 2             uSelCol = 2             uSelCol = 2
numOfMoves = 0          numOfMoves = 1          numOfMoves = 2
board(0, 0) = UNDEF     board(0, 0) = UNDEF     board(0, 0) = NOUGHT
 ....                    ...                    board(0, 1) = UNDEF
                        board(1, 2) = CROSS      ...
                        board(2, 0) = UNDEF     board(1, 2) = CROSS
                         ...                    board(2, 0) = UNDEF
                                                 ...
```

Figure 12.3: Counterexample of the trap property for the test predicate *ifT_ifF_ifF* (see Fig. 12.2)

Fig. 12.3 shows the counterexample produced by the violation of the trap property of the test predicate shown in Fig. 12.2. It is easy to see that the first state in which the test predicate is satisfied is the third one.

12.3 Combining model-based testing and runtime monitoring

In this section we explain how our approach combines model-based testing and runtime monitoring. Fig. 12.4 depicts the process we propose.

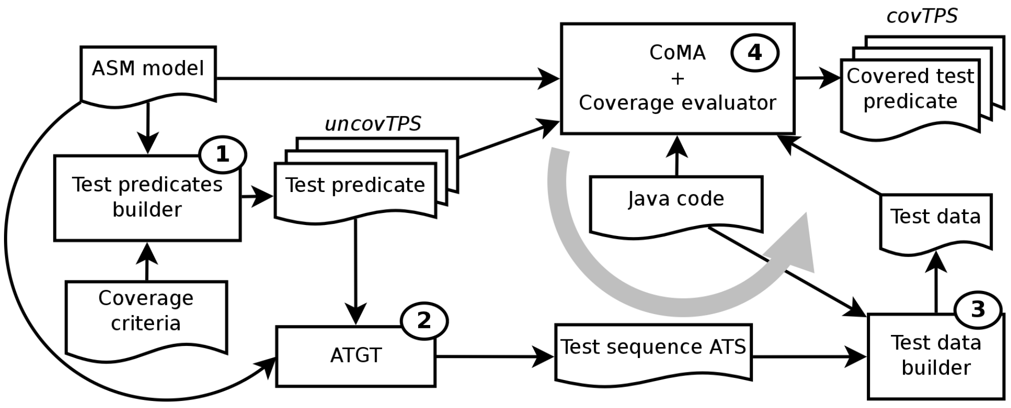


Figure 12.4: Process for testing Java programs by combining ATGT and CoMA

The process works as follows.

1. A set of test predicates *uncovTPS* is built from an ASM and a set of coverage criteria.

2. An uncovered test predicate *tp* is randomly chosen from *uncovTPS*. ATGT produces, if possible, an abstract test sequence *ATS* that covers *tp* (see Section 12.2.2). If *tp* is unfeasible, it is removed from the collection, while in case of model checker empty result (maybe because of the state space explosion problem), the process continues with another test predicate.

3. The test data builder translates *ATS* to a concrete input sequence (*test data*) for the Java implementation (see Section 12.3.1).

4. The test data are executed and runtime monitored through CoMA, which provides the oracles for the test data and evaluates the test predicate coverage (see Section 12.3.2). During the test data execution, the test predicates that are covered are removed from *uncovTPS* and added to *covTPS*, the set of covered test predicates. The process restarts from point 2 until the desired coverage is reached (see Section 12.3.3).

In our process the test data generation and the test execution are combined together: a test is executed right after it has been constructed. Such approach permits to build only the necessary test data. Sometimes this approach is called *online* (or *on-the-fly*) testing [165, 119], that distinguishes from traditional *offline* testing where a complete test suite is built before the test execution.

12.3.1 Test data construction

In Section 12.2.2 we have seen a procedure to derive, from a specification, test sequences that cover some test predicates (step 2 in Fig. 12.4).

From each abstract test sequence *ATS*, our tool derives concrete Java test data, consisting of a sequence of method calls (step 3 in Fig. 12.4). The expected outputs in the *ATS* are discarded and so the concrete tests do not contain any oracle.

The procedure that identifies the inputs in an abstract test sequence and maps them in method invocations with values for their parameters exploits the Java annotations mechanism used by CoMA to link the implementation with its abstract specification[1]:

- The value of the monitored function in the `@RunStep` annotation (e.g., `action` in the Tic-Tac-Toe example) identifies what method must be called.
- The values of the monitored functions linked in the `@Param` annotations of the (possible) method formal parameters are used as actual parameters in the method invocation (e.g., the formal parameters `r` and `c` of method `execComputerMove` are connected to the monitored functions *uSelRow* and *uSelCol*).

```
public void testIfTifFifF {
    TicTacToe t = new TicTacToe();
    t.execUserMove(1, 2);
    t.execComputerMove();
    t.execComputerMove();
}
```

Code 12.3: Test data derived from the counterexample for the test predicate *ifT_ifF_ifF* (see Fig. 12.3)

Code 12.3 shows the test data produced starting from the counterexample shown in Fig. 12.3. In each state, the value of the monitored function `action` (that is linked in the `@RunStep` annotations) is used to identify what method must be executed: if its value is `U_MOVE`, the method `execUserMove` is executed; otherwise, if its value is `C_MOVE`, the method `execComputerMove` is executed. When the method `execUserMove` must be executed, the values of the monitored functions *uSelRow* and *uSelCol*, that are linked in the `@Param` annotations of its formal parameters `r` and `c`, are used as actual parameters for `r` and `c`.

[1]Let's recall from Section 11.2.1 that the annotation `@RunStep` identifies the changing methods of the implementation. The annotation has two attributes: *setFunction* specifying the name of a monitored function of the ASM model, and *toValue* specifying the value that the function must be set to, when the corresponding method is executed. The annotation `@Param`, instead, can annotate parameters of the changing methods and of the observed constructors. It has an attribute *func* specifying the name of a monitored function of the ASM model: when the corresponding method is executed, the function is set to the value of the actual parameter.

12.3.2 Using CoMA as test oracle and coverage evaluator

In our approach we do not derive the oracle from the test sequence, as done in classical MBT for deterministic systems, but we use runtime monitoring to provide the oracle (step 4 in Fig. 12.4). If CoMA detects a not conforming behaviour during monitoring, it signals a failure.

At each Java step, CoMA also checks which test predicates are covered; note that a test predicate may be not covered by *its* test: if the implementation, due to some internal nondeterminism, chooses a different behaviour, the observed behaviour may not cover the test predicate which the test sequence is generated for. In this case, the test predicate is kept in the collection of uncovered predicates *uncovTPS*. Note, however, that a test predicate can be removed from the collection even during the execution of test cases generated for other test predicates.

12.3.3 Using test predicates as a measure of conformance

The aim of runtime monitoring techniques is to observe a system while it is running and determine if it assures some properties. Empirically, the more the system is executed and monitored, the higher is the confidence that the system is correct. But, how to measure such degree of confidence? To do this we can use coverage criteria. The idea is using CoMA not only to verify that the implementation is conformant with the specification, but also to identify what test predicates generated by MBT have been covered (step 4 in Fig. 12.4).

We introduce a *conformance index*

$$CI = \frac{\#tpsCovered}{\#tpsFeasible} \tag{12.6}$$

that provides an indication of *how deeply* a system has been monitored. *CI* could be used to decide when to interrupt the runtime monitoring: when *CI* becomes greater than a threshold P, we are confident enough that the system is correct, and we stop monitoring.

In our experiments we have used the conformance index shown in Formula 12.6. However we could use a slightly different index, defined as follows

$$CI_w = \sum_{i=1}^{n} w_i \frac{\#tpsCovered(C_i)}{\#tpsFeasible(C_i)}$$

where $C_1, \ldots, C_n$ are the considered coverage criteria, $tpsCovered(C_i)$ the covered test predicates of criterion C_i, $tpsFeasible(C_i)$ the feasible test predicates of criterion C_i, $w_i >= 0$ for each $i \in [i, n]$, and $\sum_{i=1}^{n} w_i = 1$. w_i are weights that determine the *importance* of a criterion. If all the criteria are considered equally, then $w_i = 1/n$ for each $i \in [i, n]$.

12.4 Experiments

To evaluate our approach we use the Tic-Tac-Toe as case study, previously described in Section 12.1.

We have run all the experiments on a Linux machine, Intel(R) Core(TM) i7, 4 GB Ram. In order to obtain short counterexamples, we have used the breadth-first-search option in Spin.

We consider the following coverage criteria: rule coverage, update coverage, and MCDC. Totally, we have generated 258 test predicates, 27 of which are unfeasible. For each test predicate, we have always been able either to produce a counterexample or to prove its unfeasibility, since Spin has always terminated with a not empty result.

As first experiment, we want to assess the viability of our method by applying the process described in Section 12.3 and requesting that a given percentage P of (feasible) test predicates is covered. We are interested in computing the following indicators:

1. *Conformance Index (CI)*: the percentage of feasible test predicates actually covered (see Section 12.3.3). *CI* may be greater than P because a test may cover more test predicates than requested.

P(%)	10%	20%	30%	40%	50%	60%	70%
Conformance Index (CI) (%)	19%	26%	37%	46%	57%	66%	74%
# Unfeasible	0.31	0.25	0.5	1	1.67	6	15.67
# Selected	1	1.5	3.67	6.75	12.33	28	71
# Checked and not covered	0.86	1.25	3.33	5	10.33	19.5	51
Java test length	19.43	42.25	162.33	183.5	379.33	781	2045.67
Mutation score (%)	74.58	84.60	84.99	85.39	85.56	85.70	86.21
Time (seconds)	30.49	58.49	139.14	248.79	453.66	1036.12	2412.28

Table 12.1: Experimental results

2. *Unfeasible*: the number of test predicates found unfeasible.
3. *Selected*: the number of test predicates selected, that is equal to the number of iterations in the process depicted in Fig. 12.4. Note that a test predicate could be selected more than once, in case it has not been covered by the tests previously generated for it and neither in all the other tests.
4. *Checked and not covered*: the number of test predicates for which a test has been generated but, due to some different choices in the implementation, they have not been covered by their tests and neither by other tests.
5. *Java test length*: the total number of Java statements executed in the tests.
6. *Mutation score*: to evaluate the capability of our approach to detect faults, we have applied mutation analysis by using the tool Javalanche [155]. The *mutation score* is the ratio between the faults detected over all the faults injected in the code.
7. *Time*: the time taken to complete the process.

For different values of P, we have applied our technique for 20 times and computed the average of the data. We have not been able to apply our technique with P greater or equal to 80% in a reasonable time (we put a time limit of one hour for each experiment). Table 12.1 reports the overall data.

As shown in the table, one test is enough to cover 10% of test predicates, and in this case only around 20 Java instructions are executed with an already acceptable mutation score (74%). By increasing P, all the quantities increase. The mutation score reaches a maximum around 86%. We found that some test predicates represent behaviours which have a very low probability of being executed by the implementation and, for this reason, they are not covered. The fact that the mutation score reaches a maximum, is because some injected faults are not related to the behaviour of the implementation, and special test cases should be developed for them. Finally, we can see that the *conformance index* (CI) is a good index of the fault detection capability of the monitoring activity, since it grows together with the mutation score.

Since we were already confident that the implementation actually conformed to its specification, we expected no fault in the implementation, and this was the case.

12.4.1 Comparison with other approaches

We have compared our approach with other two state-of-the-art techniques that generate test cases (with oracles), namely Evosuite and Randoop. Their results are shown in Table 12.2.

Evosuite [70] is a tool that automatically generates test cases by applying a search-based approach that generates and optimizes whole test suites, rather than generating distinct test cases directed towards distinct coverage goals. It is able to suggest oracles by adding small sets

Tool	EvoSuite				Randoop		
Options	branch coverage		mutation		100	1k	10k
	assert	no assert	assert	no assert		-	
Gen. time (sec.)	1004	154	943	605	1	5	20
Oracles	auto+mod.	No	auto+mod.	No	auto+mod.		
N. of tests	12	12	63	86	50	500	5000
Java test length	205	152	1070	1353	483	7305	87266
N. of asserts	27	N/A	125	N/A	89	1448	18618
Exec. time (sec.)	0.05	0.05	0.13	0.14	0.07	0.4	2.85
Mut. score avg (%)	29.84	30.65	54.04	26.82	39.94	45.16	70.56
Mut. score var	0.66	0.2	1.95	0.22	4.96	3.03	4.55

Table 12.2: EvoSuite and Randoop results

of assertions that summarize the current behaviour. *Randoop* [143] generates unit tests using *feedback-directed random testing*, a technique inspired by random testing that uses execution feedback gathered from executing test inputs as they are created, to avoid generating redundant and illegal inputs. It allows annotation of the source code to identify methods to be omitted and observer methods to be used for assertion generation.

Both methods are able to generate test suites together with oracles, by capturing the current behaviour of the system. This works well in order to protect against future defects breaking the current behaviour, but tends to generate *falsely failing* tests (Formula 12.3) in correspondence of nondeterminism. For this reason, we had either to modify the generated tests in order to make them pass (auto+mod. in the Table 12.2) or generating them without assertions (where possible). Note that even without assertions a test has a residual fault detection capability due to some implicit oracles (e.g., no `NullPointer` exception).

No test suite generated by Evosuite or by Randoop is able to reach the mutation score obtained by our approach. Even the best Randoop test suite has a mutation score comparable with our worst test suite, but it requires a much greater number of Java instructions and a comparable time to generate it, ignoring the time required to modify the falsely failing tests. Evosuite produces smaller tests, but with a reduced mutation score.

Part IV

Scalability issues

Dealing with memory and temporal constraints

When developing formal validation and verification techniques, one of the main problem is the scalability of the approaches. Indeed, usually the memory consumption and execution time grow exponentially with the size of the system, so that only small specifications can be analysed.

Model checking, for example, suffers from the well known *state space explosion problem* [16]; having n boolean variables in the model, the number of states is 2^n. Several techniques exist to overcome this limitation, like symbolic representation of states, compact storing of states, and efficient state space exploration. However, in model checking the dimension of the model remains a limiting factor.

Execution time problems, instead, arise when the adopted techniques use algorithms that are at least NP-complete. The problem of boolean satisfiability, i.e., discovering if a boolean formula has a model, is the most known NP-complete problem. However, new algorithms and better heuristics are constantly added to SAT/SMT solvers [153, 156], so that these tools can be efficiently used in practice.

Sometimes, when developing a new formal analysis technique, there could be some characteristics of the specifications and/or of the systems under analysis that can be exploited to address memory and temporal constraints. We show how, in two techniques for test case generation, which use model checkers and SAT/SMT solvers, the particular structure of the specifications (ASM models or boolean formulae) is exploited to provide some optimizations that permit to achieve better performances in the test generation processes.

In Chapter 13 we address the state space explosion problem for test generation from ASMs using model checkers. In Chapter 14 we propose some optimizations for the process of test generation for boolean expressions using SAT/SMT solvers.

Chapter 13

Addressing the state space explosion problem in test generation for ASMs

In Chapter 12 we have already introduced the model-based testing technique that generates test cases starting from ASM specifications. The technique derives, from *coverage criteria*, some *test predicates*, representing particular test goals. For each test predicate *tp*, the ASM specification is model checked against the trap property $\neg tp$. If a counterexample is found (i.e., the test predicate is feasible), this represents the test case to be used to cover the test predicate. Since the approach is based on model checking, its use is limited by the state space explosion problem.

In a MBT project in which we model web applications with ASMs and use the model checking approach for test case generation, we quickly encountered this problem.

However we have noticed that the system under test may have some peculiarities that can be exploited to limit the state space explosion problem. We focus on systems that are composed of independent sub-systems that pass the control to each other, such that only one sub-system is active at any time. In a web application, for instance, only one page is active at any time.

Such systems can be modeled as *sequential net*s of ASMs, defined in Section 13.1, that are sets of ASMs having some features including that only one ASM is active at every time.

In Section 13.2 we present a technique that is able to generate tests for a sequential net of ASMs, reducing the state space explosion problem. A test suite that covers every single machine is generated. These test suites are combined in order to obtain a test suite for the whole system. Under some assumptions, this technique preserves coverage of the entire system and considerably reduces the effort required to generate the whole test suite, as reported in the experiments using a benchmark example (in Section 13.3) and a simple web application (in Section 13.4).

13.1 Sequential nets of ASMs

We consider those systems that are composed of independent sub-systems that pass the control to each other, so that only one sub-system is active at any time. Usually, in order to describe such kind of systems, a model of each sub-system is developed. Moreover, a model of coordination is needed for representing the execution of the entire system, i.e., the activation/deactivation of sub-system models according to their local decisions.

A typical example is that of web applications. In a web application only one web page is active at any time, and the active page *decides* which is the next page to be displayed. The coordination is performed by the web browser and the web server that are responsible of closing the current page and visualizing the next one (passing the control among pages).

13.1.1 Definition of sequential net of ASMs

We assume that each component of the system is modeled with an ASM and we introduce the notion of sequential net of ASMs as follows.

Definition 13.1. *A* sequential net of machines *is a set of Abstract State Machines* $M_1, \ldots, M_n$ *such that:*

1. *each machine has only one initial state,*
2. *the machine M_1 is the initial machine,*
3. *only one machine is active at any time,*
4. *the active machine decides when and to which machine the control is passed,*
5. *the net is connected, i.e., each machine is reachable from the initial machine.*

A sequential net of ASMs allows one to model a set of machines that do not run in parallel, pass the control to each other, and do not share information, although they share the same environment. We call the net *sequential* because only one machine is running at any time, so the machines are not concurrent; however, there may not be an unique sequence among the machines, since every machine can decide the next machine depending on local decisions. A sequential net is a graph, where each node is a machine and an arc is a transfer of control between two machines.

A possible way to model every single machine M_i of the net, so that it can signal the transfer of control, is the following:

1. add a domain $AsmDomain = \{M_1, \ldots, M_n\}$ to its signature;
2. add a 0-ary function *currAsm* of type *AsmDomain* to its signature; *currAsm*, in the initial state, must assume the value M_i;
3. write the main rule as follows:

   ```
   if $currAsm = M_i$ then
       r_m$i$[]
   endif
   ```

   where r_mi[] is a macro rule that contains the actions of the machine.

Every machine M_i can be independently executed. It executes some useful actions until it changes the value of *currAsm*; after that any other step of execution does not produce any change in the controlled part of the machine.

Consider, for instance, the three ASMs shown in Codes 13.1, 13.2 and 13.3. They constitute a sequential net of ASMs (see Fig. 13.1). For the sake of brevity, we do not specify the internal actions of the machines.

13.1.2 Product machine

Several validation and verification activities can be directly performed on the single machines. However, if we want to do a more general evaluation of the system (e.g., simulation of the transitions among machines, or test generation for the whole system), we must also provide a model of the coordination.

One possible simple way is to merge all the machines in an unique *product* ASM as follows:

- the signatures of the machines are merged in a single signature; there is just one copy of the *AsmDomain* domain and of the *currAsm* function in the product machine;
- all macro rules (except the main rules) of the single machines are included;
- in the main rule *r_main*, rules *r_m*i[] are individually called according to the value of the function *currAsm*;
- the initial states are merged; the function *currAsm* is initialized to the value *M*1 (the first sub-system is active in the initial state).

Given the sequential net shown in Fig. 13.1, the product machine is the one shown in Code 13.4.

```
asm M1

signature:
  enum domain
    AsmDomain = {M1, M2, M3}
  monitored a: Integer
  controlled currAsm: AsmDomain

definitions:
  rule r_m1 =
    if a = 2 then
      currAsm := M2
    else if a = 5 then
      currAsm := M3
    else
      // do machine M1 actions
    endif endif

  main rule r_main1 =
    if currAsm = M1 then
      r_m1[]
    endif

default init s0:
    function currAsm = M1
```

Code 13.1: Machine M1

```
asm M2

signature:
  enum domain
    AsmDomain = {M1, M2, M3}
  monitored b: Integer
  controlled currAsm: AsmDomain

definitions:
  rule r_m2 =
    if b = 2 or b = 30 then
      currAsm := M1
    else if b = 5 or b = 100 then
      currAsm := M3
    else
      // do machine M2 actions
    endif endif

  main rule r_main2 =
    if currAsm = M2 then
      r_m2[]
    endif

default init s0:
    function currAsm = M2
```

Code 13.2: Machine M2

```
asm M3

signature:
  enum domain
    AsmDomain = {M1, M2, M3}
  monitored c: Integer
  controlled currAsm: AsmDomain

definitions:
  rule r_m3 =
    if c = 2 then
      currAsm := M2
    else if c = 5 then
      currAsm := M1
    else
      // do machine M3 actions
    endif endif

  main rule r_main3 =
    if currAsm = M3 then
      r_m3[]
    endif

default init s0:
    function currAsm = M3
```

Code 13.3: Machine M3

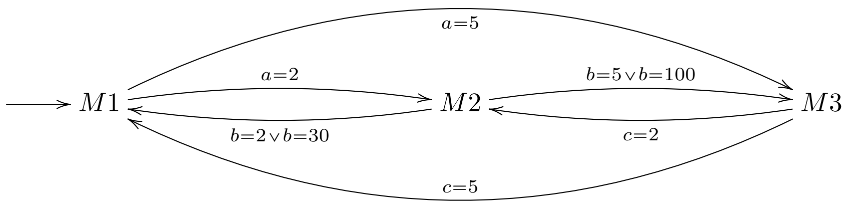

$$\begin{tikzcd}
\arrow[r] & M1 \arrow[r, bend left, "a=2"] \arrow[rr, bend left=40, "a=5"] & M2 \arrow[l, bend left, "b=2\vee b=30"] \arrow[r, bend left, "b=5\vee b=100"] & M3 \arrow[l, bend left, "c=2"] \arrow[ll, bend left=40, "c=5"]
\end{tikzcd}$$

Figure 13.1: Three ASMs (Codes 13.1, 13.2 and 13.3) constituting a sequential net

13.2 Test generation for sequential nets of ASMs

In order to efficiently test a system modeled as a sequential net of ASMs, it is not enough to test the single sub-systems, since also the interaction among them must be tested. So, we must generate test sequences that cover the whole application and not just the single sub-systems.

The first idea is to derive the test sequences directly from the product machine that already contains all the interactions among sub-systems (we call this approach *naïve*). However, since test generation algorithms based on model checking may need to visit the whole state space of the model, the generation of test sequences from the product machine may suffer from the state space explosion problem.

It would be desirable to have a method in which the model checking must be executed only on the single machines and not on the product machine; indeed, it is computationally easier to execute the model checker several times over small models, rather than executing it one time over a big model. The method should also provide a mechanism for combining the test suites produced for the single machines in an unique test suite to use for testing the whole system: the time taken by the combination of the test suites should be negligible. The proposed approach is depicted in Fig. 13.2.

```
asm ProductM

signature:
    enum domain AsmDomain = {M1, M2, M3}
    monitored a: Integer
    monitored b: Integer
    monitored c: Integer
    controlled currAsm: AsmDomain

definitions:
    rule r_m1 =
        if a = 2 then
            currAsm := M2
        else if a = 5 then
            currAsm := M3
        else
            // do machine M1 actions
        endif endif

    rule r_m2 =
        if b = 2 or b = 30 then
            currAsm := M1
        else if b = 5 or b = 100 then
            currAsm := M3
        else
            // do machine M2 actions
        endif endif

    rule r_m3 =
        if c = 2 then
            currAsm := M2
        else if c = 5 then
            currAsm := M1
        else
            // do machine M3 actions
        endif endif

    main rule r_main =
        if currAsm = M1 then
            r_m1[]
        else if currAsm = M2 then
            r_m2[]
        else
            r_m3[]
        endif endif

default init s0:
    function currAsm = M1
```

Code 13.4: Product machine of the ASMs in Codes 13.1, 13.2 and 13.3

13.2.1 Generating the test suites for every machine

We use model checking as described in Section 12.2 to generate a test suite for every ASM (step 1 in Fig. 13.2). Given the test sequences of a machine M_i, we define *inner* those sequences that terminate in a state in which *currAsm* is M_i, and *exiting* those sequences that terminate in a state in which *currAsm* is M_j (with $j \neq i$). Inner test sequences keep the control of the net in the current machine, whereas exiting sequences pass the control to another machine.

13.2.2 Building the test sequence graph

Starting from the test suites produced for the ASMs, we build a graph (step 2 in Fig. 13.2), called *test sequence graph*, where every node is a machine and every arc is a test sequence. Test sequences that do not change the current machine (i.e., *inner* sequences in the single ASM) are self loops of a node; test sequences that change the current machine (i.e., *exiting* sequences in the single ASM), instead, are arcs between different nodes.

13.2.3 Combining the tests by visiting the test sequence graph

Finally, we must build the *global test suite* (step 3 in Fig. 13.2), i.e., a test suite for the entire system. The algorithm used to visit the graph and build the *combined* test sequences of the global test suite is shown in Alg. 1.

The procedure executes a depth-first search of the graph. It takes as argument a node n to visit and a test sequence *prefix* that permits to reach n; n is marked as *visited* (line 1) in order to not be visited again and *prefix* is added to the test suite *testSuite* we are building (line 2). Then, for each exiting arc of n

- the new prefix *prefixToFn* is built concatenating the current *prefix* with *testSeq*(*arc*), i.e., the test sequence that brings to the final node of the arc (line 4);

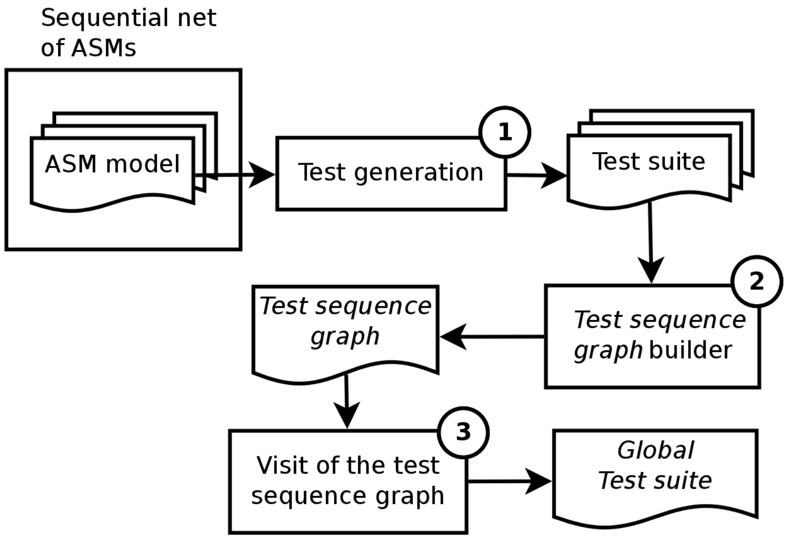


Figure 13.2: Test generation for sequential nets of ASMs

```
Require: the node $n$ to visit
Require: a test sequence $prefix$ that permits to reach the node $n$
1: $visitedNodes \leftarrow visitedNodes \cup n$
2: $testSuite \leftarrow testSuite \cup prefix$
3: for $arc \in outArcs(n)$ do
4:   $prefixToFn \leftarrow prefix + testSeq(arc)$
5:   if $finalNode(arc) \notin visitedNodes$ then
6:     $visitGraph(finalNode(arc), prefixToFn)$
7:   else
8:     $testSuite \leftarrow testSuite \cup prefixToFn$
9:   end if
10: end for
```

Algorithm 1: Visiting the test sequence graph. Procedure *visitGraph*.

- if the final node has not already been visited, it is visited using as prefix *prefixToFn* (line 6); otherwise, *prefixToFn* is added to the test suite (line 8).

The procedure *visitGraph* is invoked using as argument the initial machine M_1 of the net and the empty test sequence ϵ.

Note that the visit of the test sequence graph has linear complexity with the number of arcs and nodes and it requires a negligible amount of time with respect to the generation of the test suites.

It is straightforward to prove that the test sequences obtained with the presented algorithm are valid sequences for the product machine.

The presented visiting procedure tends to create test suites with several short tests because, when an already visited node is reached, the iterative visit is stopped and the built test is added to the test suite. In order to create smaller test suites with longer tests, in the algorithm we can modify the condition that stops the iterative visit. A first solution could be to impose a bound $K \geqslant 1$ to the number of times that a node can be visited (in the procedure proposed here, K is 1). Another solution could be to record the covered transitions (and not the states), and stopping the iteration in the states in which there are no uncovered exiting transitions.

13.2.4 Coverage

We are interested in investigating the relationship between the coverage provided by a test suite obtained from the single machines using the proposed approach, and the coverage provided by using the product machine instead (the naïve approach).

Definition 13.2. *A coverage criterion C is* preservable *if any test suite TS, obtained by the combination of tests suites $TS_1, \ldots, TS_n$ that satisfy C over the single machines $M_1, \ldots, M_n$, satisfies C over the product machine.*

If a criterion is preservable, we can satisfy it on the product machine deriving the test sequences from the single machines and combining them later. For instance, the *rule coverage*[1] criterion is *preservable* because of the following reasons:

1. by definition of *sequential net*, every machine is reachable starting from the initial machine; in each single machine M_i, every transition from M_i to another machine is specified with the update of the *currAsm* function;
2. if the *rule coverage* criterion is satisfied in every machine, it means that every rule is executed, including all the updates of the function *currAsm*. So, for each transition, there exists a test sequence that contains it;
3. by construction, the *visitGraph* algorithm assures that, if a node of the test sequence graph is reachable, a test sequence that reaches that node is built;
4. in the main rule, the product machine describes the sequential net without adding or removing any transition: at each step it simply executes the rule of the machine specified by *currAsm*.

13.2.5 Limits of the approach

The major limit of the proposed approach is that not all criteria are preservable. A criterion, in order to be preservable, must satisfy a necessary (but not sufficient) condition: it must require that, for each machine M_i (with $i \neq 1$), there exists a test sequence of another machine that reaches M_i. The rule coverage criterion satisfies such condition, since it covers all the transitions to other machines. Let's see a criterion that, since it does not satisfy such condition, is not preservable:

C_{np}: A test suite satisfies the criterion C_{np} if every macro rule r_i is fired in at least one test sequence.

Let's see the test generation process using C_{np}. Let Ma and Mb be two ASMs, shown, respectively, in Codes 13.5 and 13.6, that constitute the net shown in Figure 13.3. The product machine is shown in Code 13.7.

In the machine Ma, the criterion C_{np} is satisfied if there exists a test sequence in which the macro rule r_mA fires; C_{np} is satisfied, for example, by the test suite $TS_A = \{ts_A^1\} = \{[(gA = 0, currAsm = Ma), (gA = 0, currAsm = Ma)]\}$. In the machine Mb, C_{np} can be satisfied if there exists a test sequence in which the macro rule r_mB fires; it is satisfied, for example, by the test suite $TS_B = \{ts_B^1\} = \{[(gB = 0, currAsm = Mb), (gB = 1, currAsm = Ma)]\}$[2]. The test sequence graph obtained from test suites TS_A and TS_B is shown in Fig. 13.4.

The test suite obtained from the visit of the test sequence graph is $TS_{AB} = \left\{\left[ts_A^1\right]_{rnd(gB)}\right\} = \{[(gA = 0, gB = 345, currAsm = Ma), (gA = 0, gB = 7, currAsm = Ma)]\}$, that is the test

[1] A test suite satisfies the *rule coverage* criterion if, for every rule r_i of the ASM, there exists at least one state in a test sequence in which r_i fires and there exists at least a state in a test sequence in which r_i does not fire.

[2] Any not empty test suite (with any value for monitored functions gA and gB) satisfies the criterion over machines Ma and Mb because the execution of macro rules r_mA and r_mB does not depend on the evaluation of any guard.

```
asm Ma

signature:
    enum domain AsmDomain = {Ma, Mb}
    monitored gA: Integer
    controlled currAsm: AsmDomain

definitions:
    rule r_mA =
        if gA > 0 then
            currAsm := Mb
        endif

    main rule r_mainA =
        if currAsm = Ma then
            r_mA[]
        endif

default init s0:
    function currAsm = Ma
```

Code 13.5: Machine *Ma*

```
asm Mb

signature:
    enum domain AsmDomain = {Ma, Mb}
    monitored gB: Integer
    controlled currAsm: AsmDomain

definitions:
    rule r_mB =
        if gB = 0 then
            currAsm := Ma
        endif

    main rule r_mainB =
        if currAsm = Mb then
            r_mB[]
        endif

default init s0:
    function currAsm = Mb
```

Code 13.6: Machine *Mb*

$$\begin{tikzcd} {} \arrow[r] & Ma \arrow[r, bend left, "gA>0"] & Mb \arrow[l, bend left, "gB=0"] \end{tikzcd}$$

Figure 13.3: Sequential net of machines *Ma* and *Mb* (Codes 13.5 and 13.6)

suite produced for *Ma* where the values of *gB* are randomly chosen. In the product machine *ProductMaMb*, shown in Code 13.7, C_{np} is not satisfied using the test suite TS_{AB}, since macro rule *r_mB* never fires.

Nevertheless, it is possible to build a test suite that satisfies the criterion C_{np} in *ProductMaMb*, such as $TS_P = \{[(gA = 1, gB = 235, currAsm = Ma), (gA = 456, gB = 1, currAsm = Mb), (gA = 73, gB = 3, currAsm = Mb)]\}$.

Another limit of our approach is that the model checker may fail to find any test sequence that reaches one machine, although such sequence would be required by the (preservable) criterion. This may happen, for instance, because of the state space explosion problem in a single machine. Of course, if this case occurs, it would be even more likely that the model checker would fail on the product machine as well.

The assumption that the machines do not share information limits the applicability of our technique. It can be applied only if the different sub-systems modeled by different ASMs either do not share any information or share information that does not influence the behaviour of the machines. For instance, in a web-based application (as the case study application later introduced in Section 13.4.1) all the pages share the username (which is shown in the web pages) and the session information, which, however, do not appear in the ASMs since they do not influence the behaviour. If the web pages shared behavioural information, then our approach would not be applicable.

13.3 Initial experiment

In order to evaluate our approach, we have used a small system. It resembles the combination lock finite state machine [134], for which generating a transition covering test suite becomes exponentially expensive. The problem is that of discovering the key of an electronic combination lock made of n digits having values from 1 to x. We have modeled the system as a sequential net of ASMs (see Fig. 13.5). The net is composed of n machines; every machine M_i has a monitored

```
asm ProductMaMb

signature:
    enum domain AsmDomain = {Ma, Mb}
    monitored gA: Integer
    monitored gB: Integer
    controlled currAsm: AsmDomain

definitions:
    rule r_mA =
        if gA > 0 then
            currAsm := Mb
        endif

    rule r_mB =
        if gB = 0 then
            currAsm := Ma
        endif

    main rule r_main =
        if currAsm = Ma then
            r_mA[]
        else
            r_mB[]
        endif

default init s0:
    function currAsm = Ma
```

Code 13.7: Product machine of the machines *Ma* and *Mb* (Codes 13.5 and 13.6)

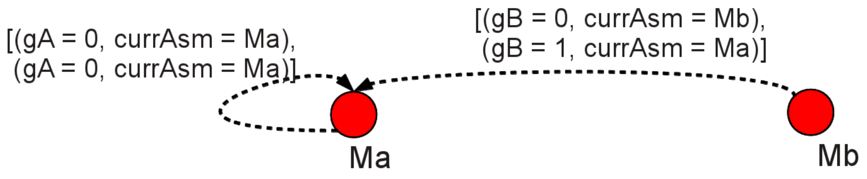


Figure 13.4: Test sequence graph obtained with the criterion C_{np} over Ma and Mb (Codes 13.5 and 13.6)

function a_i in the range $[1, x]$. If a_i (with $i = 1, \ldots, n-1$) takes the specific value 1, then the next machine M_{i+1} becomes active; if a_j (with $j = 2, \ldots, n$) becomes greater than $x/2$ then the system goes back to machine M_1, otherwise the machine M_j remains active.

We have evaluated our method depending on the number of digits (machines) n and/or the base x (the cardinality of the codomain of functions a_i).

For each combination of n and x we have built n single machines, where each machine has nx states since the signature of each machine M_i is composed of two 0-ary functions, a_i and $currAsm$, whose codomain sizes are, respectively, x and n. Then we have built the unique product machine that has nx^n states, since there are n 0-ary functions whose codomain size is x, and a 0-ary function whose codomain size is n.

Then we have generated the test sequences both for the product machine (naïve approach) and for the sequential net of machines by the method introduced in Section 13.2; all the coverage criteria introduced in [76] have been used. As expected, we discovered that it is easier to execute n times the model checker over the single machines rather than executing the model checker one time over the product machine. The results of the experiment are shown in Fig. 13.6; the

$$\begin{tikzcd}
M_1 \arrow[r, "a_1=1"] & M_2 \arrow[r, "a_2=1"] \arrow[l, bend right, "a_2>x/2"'] & M_3 \arrow[r, "a_3=1"] \arrow[ll, bend left, "a_3>x/2"] & \cdots M_i \cdots \arrow[r, "a_{n-1}=1"] \arrow[lll, bend right=50, "a_i>x/2"'] & M_n \arrow[llll, bend left=40, "a_n>x/2"]
\end{tikzcd}$$

Figure 13.5: Combination lock problem — Sequential net of ASMs

dependence between the execution time and the number of single machines n is reported.

If the single machines are used, the execution time grows linearly with the number of machines; if the product machine is used, instead, the execution time grows exponentially with the number of machines. We made several experiments with different values for x (the cardinality of the codomain of functions a_i); as expected, in the product machine the value of x influences the execution time (even for small changes of x), whereas in the single machines it is irrelevant. We report the experiments made with the product machine with x equal to 10, 20 and 50, and the experiment made with the single machines with x equal to 50. We set a time limit of 1 hour for each experiment setting. All the experiments were executed on a Linux PC with 8 Intel(R) Xeon(R) CPUs E5430 @ 2.66GHz and 8 GB of RAM.

Test suite sizes In Table 13.1 we report the sizes of the test suites obtained using the sequential net method and the product machine method (naïve approach). We report the sizes obtained with an increasing number of machines; we do not report the value of x because it does not influence the test suite size.

# ASMs	1	2	3	4	5	6
Sequential net	3	5	7	9	11	13
Product machine (naïve approach)	3	7	11	15	19	n/a

Table 13.1: Combination lock problem – Test suite size

From our experiments it seems that the test suites derived from the test sequence graph are

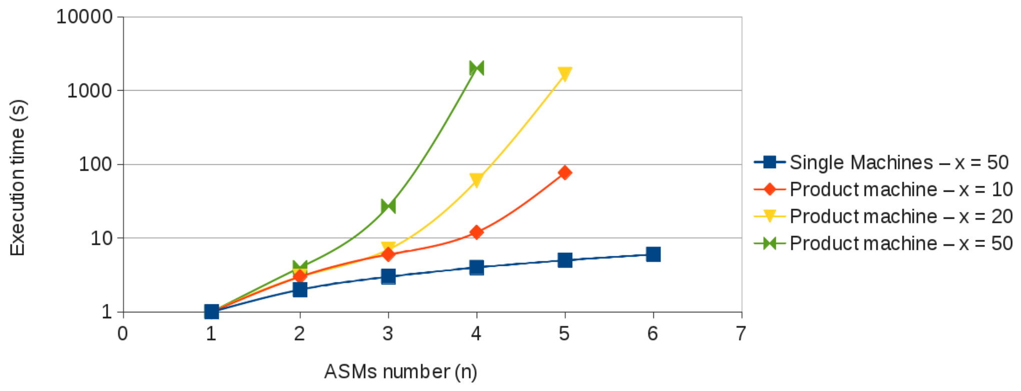


Figure 13.6: Combination lock problem – Model checker executions times (sec.)

smaller than those obtained directly from the product machine. However, we must notice that this can not be taken as a general law; additional experiments are needed to define more clearly the relationship between the sizes of the test suites obtained with the two methods.

Code coverage As sanity check, we also measured the code coverage obtained by using the two methods. We implemented the system, previously specified in ASM, into Java and translated the test suites in JUnit. We obtained the same code (statement and branch) coverage by using both the test sequences generated from the product ASM and from the sequential net.

13.4 Model-based testing of web-based applications

We have studied the test generation for sequential nets of ASMs in the context of MBT of web-based applications [59]. In this context, every machine represents a single page of the application. So, an ASM is created for each web page of the web application; in this scenario the *AsmDomain* can be interpreted as the set of web pages and the *currAsm* function as the current active page. The methodology introduced in Section 13.2 is applied to obtain a test suite for the whole web application; finally, each test sequence can be mapped to a SAHI script [152] to exercise the tests directly on the web application. In Section 13.4.1 we introduce the web application case study, and in Section 13.4.2 the application of the proposed approach for test case generation.

13.4.1 Description of the web application case study

We describe the web application case study taken from [130] we used in our experiments. There are six php pages in the web application under test and each of them, as well as their corresponding ASMs, is described below.

- `index.php` – It serves as the login interface for the website. A user is required to enter a username and a password in order to access the other three pages of the site. The *Reset* button clears all text entries, while the *Submit* button opens up `main.php`, as long as the identification credentials are correct. If any information is missing, an error message page is displayed.
- `error_b.php` – It is activated from `index.php` if any information is missing, or username or password are wrong.
- `main.php` – It permits users to execute different actions. Specifically, users can click on a link (at top left corner of page), upload a file by clicking on the *Browse* button, enter text into a textbox, select a checkbox, and click on a *Submit* button which loads `random.php`.
- `error_a.php` – It is displayed if any information is missing in `main.php`.
- `random.php` – It permits users to execute actions not available in `main.php`. Two links bring the user back to `index.php` and `main.php`. There are also drop-down lists, radio buttons, and a *Submit* button which loads `end.php`.
- `end.php` – It serves as the *end* of the web application. The user has the option of closing the web browser, or clicking on a link to return to `index.php`.

13.4.2 Test case generation

Modeling every page with an ASM At first, we modeled the complete web application with an unique ASM, i.e., using the naïve approach. The model construction was feasible but the model checking was not able to complete the test generation. So, we modeled the web application using a sequential net of ASMs where every page is represented by an ASM and the domain *AsmDomain* is composed by the web pages. The obtained sequential net is shown in Fig. 13.7.

For translating a web page behaviour into an ASM, we have put on a table the inputs of the web page (e.g., the values of the text fields) and identified, for every combination of inputs, a transition to another page or a set of state updates (i.e., a modification of the page content). In this way we have built an ASM for each web page.

Test generation For the test generation we have used, as described in Section 13.2.1, the ATGT tool over each ASM, using as coverage criteria all those described in [76].

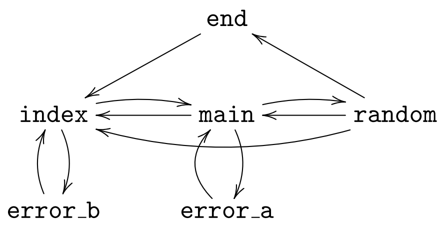


Figure 13.7: Web-based application case study – Sequential net of ASMs

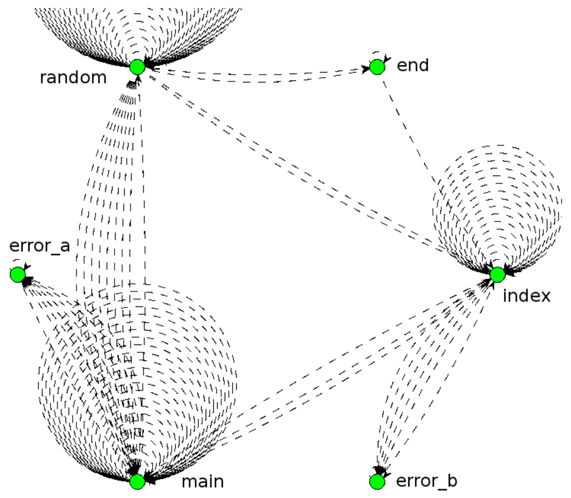


Figure 13.8: Web-based application case study – Test sequence graph

Test sequence graph construction Then we have built the test sequence graph (see Fig. 13.8) as described in Section 13.2.2. Each transition of the sequential net has been covered in the test sequence graph.

Table 13.2 reports, for each ASM, the number of test sequences, divided between *inner* and *exiting*.

	`index`	`error_b`	`main`	`error_a`	`random`	`end`
# tests	24	3	36	3	45	2
# inner - # exiting	18 - 6	1 - 2	26 - 10	1 - 2	32 -13	1 - 1

Table 13.2: Web-based application case study – Test sequences number

Test sequence combination Then, we have applied the technique presented in Section 13.2.3 in order to obtain a single test suite for the whole web application. The obtained test suite contains 212 test sequences and it satisfies all the coverage criteria used to generate the test suites over the single machines.

Test of the web application Finally, each test sequence of the test suite has been automatically mapped to a SAHI script; the execution of all the scripts has permitted us to test all the aspects of the web application. Code 13.8 shows one of the produced SAHI scripts.

```
_navigateTo("index.php");
_setValue(_textbox("username"),"admin");
_setValue(_textbox("password"),"pwd");
_click(_submit("submit"));
_click(_checkbox("agree"));
_setValue(_textarea("text"),"someText");
```

Code 13.8: Web-based application case study – SAHI script example

Chapter 14

Improving the test generation process for Boolean expressions

Boolean expression testing plays an important role in model-based testing. Boolean expressions frequently occur in complex conditions under which some specification actions are performed. In ASMs, for example, boolean expressions can be used as guards of conditional, forall, and choose rules. Coverage criteria for ASMs [76], described in Section 12.2, may require to deeply test the boolean expressions that appear as guards in the model.

It is widely known [110, 167] that a Boolean expression may be affected by certain types of errors known as *fault classes* of the expression. Exhaustive testing of Boolean conditions is not feasible in practice, since a Boolean expression f with n variables requires 2^n test cases and this n is normally very big. Therefore, testing criteria are usually applied to select only subsets of all possible test cases having, obviously, a reduced fault detection capability. Traditional approaches build a test suite from the syntactical structure of the Boolean expression. A limitation of these approaches is that they do not explicitly consider the expression fault classes. They also require expressions to be in a particular normal (usually disjunctive) form.

To overcome these limitations of traditional algorithmic testing generation methods, recent results [72] show how it is possible to reduce the problem of finding fault detecting test cases for Boolean expressions to a logical satisfiability problem, which can be solved by a SAT/SMT-based algorithm. This approach has several advantages:

a) it does not require the specifications under test to be expressed in a particular normal form, so avoiding possible overhead due to the formula transformation;

b) it generates test cases directly targeting specific fault classes;

c) it uses several reduction policies to minimize the size of resulting test suites.

The process of automatic test generation by SAT/SMT techniques, however, requires more time and memory than standard generation algorithms and this fact limits its use in practice. The contribution of this work is to improve this process by proposing a number of optimizations that promise to make SAT/SMT techniques as efficient as standard methods for test generation purposes with the mentioned benefits.

Note that SAT solvers are increasingly used for solving practical problems where one needs to satisfy several potentially conflicting constraints, and satisfiability solvers can now be effectively deployed in practical applications [128].

Also SMT solvers are increasingly used in applications. Although they are far more complex tools than SAT solvers, we here consider also the use of SMT solvers for several reasons. SMT solvers should be as powerful as SAT solvers when applied to satisfiability problems, with a minimum overhead. While most SAT solvers take as input only formulas in conjunctive normal form (CNF), SMT solvers accept as input generic Boolean formula, allowing some pre-transformations of the predicates in order to increase their efficiency. Moreover, most SMT solvers have a richer command interface, allowing, for instance, adding and retracting assertions, and this can be exploited to optimize the test generation process.

We here propose a broad range of optimizations. Some optimizations regard the actual use of the tools (e.g., avoiding exchanging files with the external solver and using native libraries instead). Other optimizations improve the SAT/SMT-based process of automatic test generation, independently from a specific input specification and selected testing criterion. Others are specific to the process instantiated for testing Boolean expressions.

The most powerful optimizations regard the *collecting process*, which proved to be most effective in building compact test suites [72].

We also propose a comparison of different SAT and SMT solvers that can be used in the test generation process and that are able to support (not necessarily all) the proposed optimizations. On the base of the best (among those used) tool, we show evidence that the proposed optimizations are effective. And, on the base of our experiments, we conclude that SMT-solvers may have better performance than SAT solvers.

Overall, we show that SAT/SMT solvers can be successfully applied to Boolean testing and that a well engineered process for test generation based on SMT solvers is capable of generating the tests for complex fault based criteria in a reasonable time without losing the advantages of the approach. We claim that the proposed optimized test generation method might be applied to other testing approaches (e.g., constrained combinatorial interaction testing [52]).

The chapter is organized as follows. The general process of test generation by using SAT/SMT solvers is described in Section 14.1, and its instantiation for testing Boolean expressions is introduced in Section 14.2. Process optimizations are presented in Section 14.3. Experimental results which bring evidence of the improvements due to the proposed optimizations are reported in Section 14.4 where several experiments, conducted by means of different SAT/SMT solvers and on diverse Boolean specifications, are presented.

14.1 Model-based test generation by SAT and SMT

In [38, 72], an approach is presented for model-based generation of tests by means of SAT and SMT solvers. This automatic test generation process is described here, recalling some basic definitions already given for the test case generation using model checkers. We assume that the input space is *complete*, i.e., every input can take any value in its domain.

Definition 14.1. *A* testing criterion TC *is a function that, given a specification S, returns a set of predicates which must be satisfied (or covered). Each predicate represents a test goal and is called* test predicate*.*

Definition 14.2. *We say that a* test t satisfies *or* covers *a test predicate tp iff t is a model of tp, i.e., $t \models tp$. A test predicate tp is said* infeasible *if there is no test that satisfies it, i.e., $\not\models tp$.*

Definition 14.3. *Given a testing criterion TC and a specification S, we say that a test suite TS is* adequate *to test S according to TC, iff*

$$\forall tp \in TC(S)\ [(\exists t \in TS\ t \models tp) \vee \not\models tp]$$

Upon the assumption of having an algorithmic way for generating the test predicates given a specification and a testing criterion, to discover if a test predicate is infeasible or to find a *test* that covers it, a SAT/SMT solver can be used (assuming that the solver terminates, otherwise it is not known whether the test predicate is feasible or not).

The naïve approach for obtaining an adequate test suite (according to Def. 14.3) requires to generate a test for each test predicate $tp \in TC(S)$ or discovering that the tp is infeasible. This naïve approach requires a high amount of time and generates huge test suites. It can be improved by several techniques: here we only consider collecting, monitoring, and post reduction[1]. The

[1]The *order* in which the test predicates are considered may impact the size of the final test suite. Some heuristics (e.g., *subsuming order* in [72]) aim at providing orders that minimize the size of the test suite. Here we do not apply any optimization to the test predicates order and we assume random ordering.

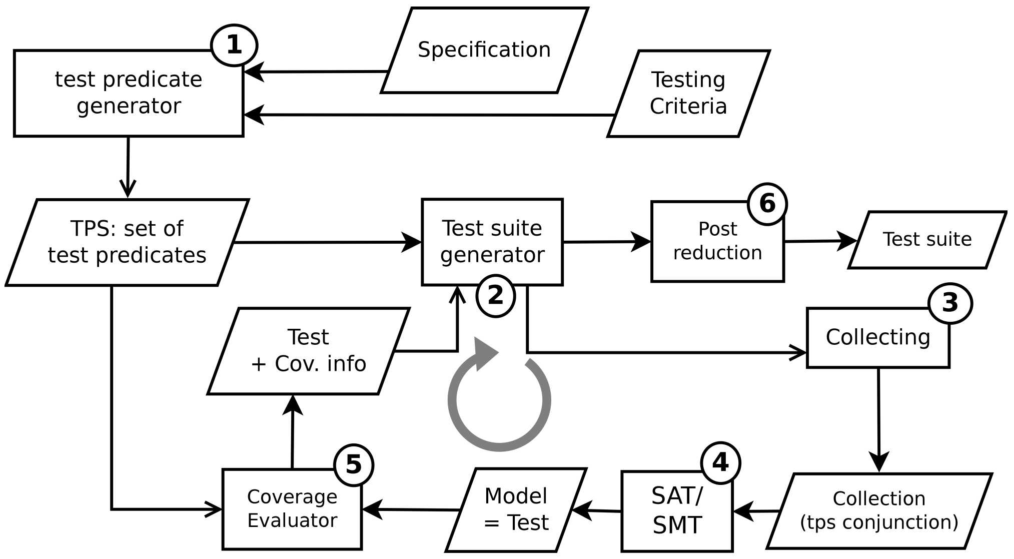


Figure 14.1: Test generation process through SAT/SMT solvers

overall test generation process by SAT/SMT solvers is depicted in Fig. 14.1.

Given some testing criteria, a set *TPS* of test predicates is generated from the specification (step 1 in Fig. 14.1) according to the Def. 14.1. Then, the process of *test suite generation* (step 2 in Fig. 14.1) is iteratively executed until, for each feasible test predicate, a model is found. The three steps of this iterative process are described in Section 14.1.1, while the last step of the process is described in Section 14.1.2.

14.1.1 Test suite generator

The test suite generation process is composed of three steps: collecting, model computation, and coverage evaluation.

Collecting A core optimization employed in this work consists in finding tests that cover as many test predicates as possible instead of single test predicates. We call *collection* the set of test predicates sharing the same model, and *collecting* the process of grouping test predicates. The collecting phase is the third step in Fig. 14.1.

From a theoretical point of view, the problem of collecting consists in partitioning the set of feasible test predicates with the minimal number of partition classes. Each class contains only test predicates sharing the same model. The number of possible partitions of a set is given by the Bell number B_n which grows exponentially[2] with n. For this reason, in order to keep the search of the optimal solution still feasible in practice, we accomplish such partition by using the greedy algorithm reported in Alg. 2 and explained in the following.

The algorithm starts with an empty collection C (line 1). Then, it tries to add every test predicate *tp* to C. This is possible only if *tp* is compatible with the other already collected test predicates. A SAT/SMT solver is called to the purpose of discovering if there exits a common model for *tp* and all the other collected test predicates by trying to `sat`isfy their conjunction (line 3). If the test predicate is not compatible, it is checked if it is satisfiable (line 6); if it is unsatisfiable, it is removed from *TPS* in order to avoid trying to collect a test predicate which is actually infeasible. Note that infeasible test predicates consume computing resources without producing usable tests.

The collecting process is very expensive in terms of solver calls, but it is able to produce

[2]Asymptotic estimates giving the following lower and upper limits are known: $2^n \leqslant Bell(n) \leqslant n! \leqslant 2^{n \log n}$

```
Require: $TPS$ : set of all the test predicates to be considered
 1: $C \leftarrow \{\}$
 2: for $tp \in TPS$ do
 3:   if $\mathtt{sat}(\bigwedge_{c \in C} c \wedge tp)$ then
 4:     $C \leftarrow C \cup \{tp\}$
 5:     $TPS \leftarrow TPS \backslash \{tp\}$ $\{tp$ is collected$\}$
 6:   else if $\mathtt{sat}(tp)$ then
 7:     $\{tp$ is feasible but it cannot be collected in $C\}$
 8:   else
 9:     $TPS \leftarrow TPS \backslash \{tp\}$ $\{tp$ is infeasible$\}$
10:   end if
11: end for
12: return $tpc = \bigwedge_{c \in C} c$
```

Algorithm 2: Collecting process

very compact test suites [72]. An example of application of the Alg. 2 for a Boolean formula is reported in Example 14.1.

Model computation Once a collection *tpc* is built, the SAT/SMT solver is invoked (step 4 in Fig. 14.1) to find a model for *tpc* and, therefore, for all the test predicates collected in *tpc*.

Coverage evaluation The *coverage evaluation* is then performed (step 5 in Fig. 14.1) on the newly generated model to check if it also covers other test predicates among those previously covered by other models or those that have not yet been collected and are not known to be unsatisfiable. This technique is also called *monitoring*. Note that, thanks to monitoring, a test predicate can be covered by more than one model.

14.1.2 Post reduction

The resulting test suite may contain tests which are not *necessary*, i.e., removing them from the test suite will lead to all test predicates still being covered (as in [91]). The problem of finding the optimal subset of the original test suite that still covers all the test goals is NP-hard, but can be efficiently solved by a simple greedy heuristic [47]. *Post reduction* (step 6 in Fig. 14.1) is the last step of the process; it removes unnecessary test predicates (if any) and it can be performed in a negligible amount of time.

14.2 Test generation process for Boolean expressions

We here assume that the specifications under test are Boolean expressions. Operands are *atomic* Boolean terms, i.e., atomic because they cannot be further decomposed in simpler Boolean expressions. We call the operands *inputs* or *variables* and use symbols like $x_1, x_2, \ldots$. The occurrence of an input in a expression is referred to as a *condition*. For example, the formula $x_1 \wedge x_2 \vee x_1$ contains two variables (x_1 and x_2) and three conditions (two x_1's and one x_2). If the expression is not normalized, i.e., no restrictions exist on how operators and conditions are joined together, we say that the expression is in a *general form* (GF). In this work we consider GF Boolean expressions. In the context of testing Boolean predicates, a *test case* is a value assignment to every Boolean variable in the formula. A *test suite* is a set of test cases.

When the specifications under test are Boolean expressions, a testing criterion TC is represented by a function that given a Boolean expression φ returns a set of predicates which must be covered (i.e., satisfied).

There exist several ways proposed in literature on how to determine the set of test predicates that must be covered in order to test φ. In the following, we briefly present the technique described in [72] that generates test predicates from GF Boolean expressions whose tests are guaranteed to detect faults in specific fault classes (as in [120, 109]).

Fault-based testing criteria Given a Boolean specification φ and a fault class F, we can easily derive all the possible faulty implementations φ'_i of φ. Several fault classes for Boolean expressions have been defined and a hierarchy among them has been established [110]. According to the faulty-based testing criteria approach, given an expression φ and ranging over all the fault classes, the testing criteria compute all the *test predicates* $tp_i = \varphi \oplus \varphi'_i$ (called *detection conditions*), where $\oplus$ denotes the exclusive or (xor). Finding the tests t_i of φ means computing models of the test predicates tp_i, i.e., $t_i \models tp_i$.

The process in Fig. 14.1 is still valid for Boolean expressions, by reading "fault classes" in place of "testing criteria", and "Boolean expression" in place of "specification". Below, we report an example of running the collecting process in Alg. 2 for a Boolean expression.

Example 14.1. Let be $\varphi = a \vee (a \wedge b)$ and $TPS = \{(a \vee (a \wedge b)) \oplus (a \wedge b), (a \vee (a \wedge b)) \oplus (\neg a \vee (a \wedge b)), (a \vee (a \wedge b)) \oplus (a \vee (\neg a \wedge b)), (a \vee (a \wedge b)) \oplus (a \vee (a \wedge \neg b))\}$ obtained by using various fault classes. Alg. 2 collects the first two test predicates since their conjunction has a model. The third test predicate can not be added to the collection because it is not compatible; nonetheless, it is feasible, so it is not removed from *TPS* and it will be considered in subsequent runs of the collecting algorithm. The fourth test predicate is infeasible and it is removed from *TPS*.

14.3 Optimizations

In this section we present the optimizations we have devised in the test generation process. Some optimizations are purely technological since they regard the use of the solvers, like avoiding the use of files (O.4). Others exploit some logical equivalences to simplify the test predicates, like O.1. Most of them refer to optimization of the collecting process: how to speed up the collecting of test predicates (those presented in Section 14.3.1) and how to quit the collecting in advance (those presented in Section 14.3.2).

O.1 *Simplification of the test predicate*

Considering that our test predicates have form $\varphi \oplus \varphi'$ and that φ and φ' often have a common subexpression, it may be useful to apply some kind of simplification of a test predicate before running the solver in order to reduce the number of conditions (i.e., occurrences of literals). We have used the following two equivalences that allow to factor a part of the formula and to push the $\oplus$ operator near the literals: Let a, b, and c be any predicate:

$$(a \wedge b) \oplus (a \wedge c) \equiv a \wedge (b \oplus c)$$
$$(a \vee b) \oplus (a \vee c) \equiv \neg a \wedge (b \oplus c)$$

Example 14.2. Consider for instance the expression $\varphi = (x_1 \wedge x_2) \vee x_3$ and apply the negation fault [72] to x_3 obtaining $\varphi' = (x_1 \wedge x_2) \vee \neg x_3$. The test predicate $\varphi \oplus \varphi'$ becomes:

$$((x_1 \wedge x_2) \vee x_3) \oplus ((x_1 \wedge x_2) \vee \neg x_3) \equiv \neg(x_1 \wedge x_2) \wedge (x_3 \oplus \neg x_3)$$

While the original test predicate has 6 conditions, the simplified version contains only 4 conditions.

O.2 *Optimizing the transformation to CNF*

Almost all SAT solvers require CNF input formulas, while Boolean expressions we consider and their test predicates have *general* form. Efficient transformation to CNF is still a research topic [150]. There are at least two classical possible alternatives: one that preserves equivalence and consists in applying several logical equivalences (double negative law, De Morgan's laws, distributive law), and the classical transformation proposed by Tseitin [163] that preserves satisfiability, avoids the size explosion of the resulting CNF, but introduces a linear number of new variables.

```
1: $C \leftarrow \{\}$
2: **for** $tp \in TPS$ **do**
3:   **if** $\mathtt{sat}(\bigwedge_{c \in C} c \wedge tp)$ **then**
4:     $C \leftarrow C \cup \{tp\}$
5:     $TPS \leftarrow TPS \setminus \{tp\}$ $\{tp$ is collected$\}$
6:   **else if** $tp \notin TPS\_FEASIBLE$ **then**
7:     **if** $\mathtt{sat}(tp)$ **then**
8:       $TPS\_FEASIBLE \leftarrow TPS\_FEASIBLE \cup \{tp\}$ $\{tp$ is feasible but not collectable in $C\}$
9:     **else**
10:      $TPS \leftarrow TPS \setminus \{tp\}$ $\{tp$ is infeasible$\}$
11:    **end if**
12:  **else**
13:    $\{tp$ is known to be feasible but it cannot be collected$\}$
14:  **end if**
15: **end for**
16: **return** $tpc = \bigwedge_{c \in C} c$
```

Algorithm 3: Collecting process with feasible predicates recording

O.3 *Avoiding the transformation to clausal form*

As argued by Jain and Clarke [101], converting a non-clausal formula to CNF requires a great effort (it can grow exponentially in length) and it may destroy the initial structure of the formula, which could be used for efficient satisfiability checking. SAT/SMT solvers taking Boolean expressions in general forms (GF) may perform better.

O.4 *Using the API and avoiding the exchange of files*

A simple optimization regards the way the solvers are invoked. The previous version of our prototype tool runs solvers by command line interface. Each invocation of the solver is done by creating a new external process at every time and the interaction with it is performed by means of files and command line strings. That solution requires that the solver is fed with input files; in this way the read/write speeds of the hard disk can increase the time taken just to invoke the tool. Since the SAT/SMT solver is repetitively called when collecting, the number of invocations of the solver rapidly increases and the tool invocation time becomes a critical factor. A simple yet critical optimization consists in avoiding this use and embedding the solver in the process itself.

14.3.1 Collecting optimizations

Since collecting (see Alg. 2) is the most powerful technique to generate small test suites, but it is also the most expensive [72], a great effort should be spent to improve this part of the generation process. In this section, we devise several techniques to speed up the collecting process.

O.5 *Marking feasible test predicates*

With respect to Alg. 2, a first optimization consists in marking not only if a test predicate is infeasible but also if it is *feasible*. Therefore, every test predicate in *TPS* can be either marked infeasible and removed from *TPS*, or marked feasible and added to *TPS_FEASIBLE*. The modified version of the algorithm is reported in Alg. 3. When a *tp* is incompatible with the other test predicates, if it is not already known to be feasible (line 6 in Alg. 3), it is checked if it is feasible by calling the solver (line 7): if *tp* is feasible, it is added to *TPS_FEASIBLE* (line 8), otherwise it is removed from *TPS* (line 10). The set *TPS_FEASIBLE* is initialized to the empty set before the iterative test suite generation process begins (right before step 2 in Fig 14.1).

O.6 *Collection with witness*

The collection algorithm (both the versions in Alg. 2 and 3) returns the collected test predicates and the SAT/SMT solver is called after the collecting process to find a model for the conjunction of the collected test predicates (step 4 in Fig. 14.1). Since the solver has been already invoked to check if the last test predicate added to the collection is compatible with the other test predicates previously collected, a further call to the SMT solver is useless. The collecting algorithm can be modified in a way that it returns the collected test predicates together with the model found for the collection. This model is a *witness*, since it is the proof that the collected test predicates can be actually collected together.

O.7 *Checking if the witness is a model*

When trying to add the current test predicate *tp* to the collection C (line 3 in Alg. 3), one could check if the witness for the collection C is already a model for *tp*. In this case *tp* can be added to C without any further call of the SAT/SMT solver.

O.8 *Collecting incrementally*

Most modern SAT/SMT solvers maintain the logical context of a given problem and allow incremental satisfiability checking. Although this concept is not uniquely defined, it means that it is possible to add expressions incrementally to the current context[3] and that satisfiability is checked after each addition. If the expression is a conjunction $e_1 \wedge \cdots \wedge e_n$, the `sat` predicate can be computed in the following way:

```
sat($e_1 \wedge \ldots \wedge e_n$) =
    for i ∈ [1 .. n]  do
        add($e_i$);
        if !solve()
            return false;
        end if
    end for
    return true;
```

In SAT solvers that requires the formulas to be in CNF, each e_i must be a clause and `add`(e_i) must be `addClause`(e_i).

O.9 *Collecting incrementally with backtracking*

Besides the incremental satisfiability, most SMT solvers, like Yices [60] and Z3 [55], have the further feature of removing an added formula from the context when this becomes inconsistent. Therefore, SMT solvers allow incremental collecting (O.8), but also backtracking of assertions.

Normally, to prove that a collection of test predicates has a model, we build a new context and we search for a model of the formula $\bigwedge_{c \in C} c \wedge tp$ (line 3 in Alg. 3). Thanks to the capability of adding and removing assertions to the context, by the Alg. 4, we can incrementally collect all the test predicates having a common model. The Alg. 4 exploits the SMT operations `push` saving the current context and `pop` restoring the previous saved context. Before adding a single candidate test predicate *tp* to the collection, the context is saved with a `push` statement, then *tp* is added by an `assert` instruction. If the context has still a model (`solve` returns true), then *tp* is added to the collection, otherwise the context is restored by a `pop`.

O.10 *Double incremental collecting*

[3]For instance, MiniSAT [63] provides the method `addClause`, Yices [60] and Z3 [55] have the instruction `assert`.

```
$C \leftarrow \{\}$
**for** $tp \in TPS$ **do**
  push {save current context}
  assert $tp$ {add $tp$ to the current context}
  **if** solve() **then**
    $C \leftarrow C \cup \{tp\}$;
    $TPS \leftarrow TPS \setminus \{tp\}$
  **else**
    {check if $tp$ is unfeasible}
    **if** $tp \notin TPS\_FEASIBLE$ **then**
      **if** sat($tp$) **then**
        ...
      **end if**
    **end if**
    pop {restore previous context}
  **end if**
**end for**
```

Algorithm 4: Incrementally collecting

In case all the test predicates have the form $\varphi \oplus \varphi'_i$ (if the $\oplus$ is not pushed), one can use the following *Xor elimination* logical equivalence:

$$\bigwedge_{i=1}^{n} (\varphi \oplus \varphi'_i) \equiv \left(\varphi \wedge \bigwedge_{i=1}^{n} \neg\varphi'_i\right) \vee \left(\neg\varphi \wedge \bigwedge_{i=1}^{n} \varphi'_i\right)$$

to simplify the collecting process. Thanks to this equivalence, one can start with two contexts: $\mathtt{c}_\top$ initially containing only φ, and $\mathtt{c}_\bot$ containing $\neg\varphi$. When a test predicate $tp_i = \varphi \oplus \varphi'_i$ must be checked for compatibility with all the test predicates already collected, $\neg\varphi'_i$ is added to $\mathtt{c}_\top$ (if still valid), while φ'_i is added to $\mathtt{c}_\bot$(if still valid). We can distinguish the following three cases:

(1) if both contexts are still satisfiable, then tp_i is accepted;

(2) if only one context is satisfiable, then tp_i is still accepted but the context without model is invalidated and no longer considered for the collection until the next new collection;

(3) if no valid context is satisfiable, then tp_i is refused and the valid contexts are restored.

14.3.2 Limiting collecting

To make the collecting process of test predicates faster, another approach consists in limiting the test predicates that can be possibly collected. In this case, instead of reducing the time necessary to collect every possible test predicate, one could try to limit the number of test predicates that are collected. The collection could not contain all the uncovered test predicates which can be possibly collected together (we can say that it is a *partial* collection), and this may reduce the effectiveness of the collecting process itself. However, this negative effect could be reduced in the coverage evaluation phase (step 5 in Fig. 14.1): if the test generated for a collection also covers test predicates which could have been collected together, then these test predicates are marked as covered and no longer considered.

O.11 *Quit after N*

A first limitation is about the maximum *number* of test predicates to be possibly added to a collection. Once that the collection contains N test predicates, the collecting process stops. This policy is very easy to implement. With very small N, it makes the collecting process very fast, but it may produce bigger test suites. By increasing N, it behaves similarly to the unlimited collection, but also the time may increase.

O.12 *Collecting until useful*

Another policy consists in performing the collecting until it is useful, but stopping it as soon as it becomes useless. Indeed, when the collected test predicates have only one unique model, collecting could be stopped without losing anything: any new test predicate that would be added to the collection would be in any case covered, in the monitoring phase (step 5 in Fig. 14.1), by the test produced by the SAT/SMT solver for the collection. We devise the following technique in order to discover if a model of a predicate is unique.

Let *asExpr* be a function that given a model m returns a Boolean predicate having m as unique model. The simplest *asExpr* is the function that returns the conjunction of the variables having value *true* in m and the negation of the variables having value *false* in m.

Example 14.3. If $m = \{\mathrm{a} = \mathit{true}, \mathrm{b} = \mathit{true}, \mathrm{c} = \mathit{false}\}$ is the model, then $\mathit{asExpr}(m) = a \wedge b \wedge \neg c$

The following proposition indicates how to check if a model of a Boolean predicate is unique.

Proposition 14.1. *Let ψ be a feasible predicate and m be a model of ψ. m is the unique model of ψ if $\psi \wedge \neg \mathit{asExpr}(m)$ is not satisfiable.*

We can use Prop. 14.1 to check if the model of the collection is unique right after a *tp* is added to the collection. If the model is unique, the collection process can be stopped.

Example 14.4. Let $\mathit{tpc} = ((a \vee b) \oplus (a \vee \neg b)) \wedge ((a \vee b) \oplus (\neg a \vee b))$ be a collection of test predicates and $\mathrm{m} = \{\mathrm{a} = \mathit{false}, \mathrm{b} = \mathit{false}\}$ be one of its models. The predicate $\mathit{tpc} \wedge \neg \mathit{asExpr}(m) = ((a \vee b) \oplus (a \vee \neg b)) \wedge ((a \vee b) \oplus (\neg a \vee b)) \wedge \neg(\neg a \wedge \neg b)$ is unfeasible, therefore m is the unique model of *tpc*.

Note that while optimization O.11 may produce bigger test suites because it may leave out of the collection some uncovered test predicates that should be collected, this optimization collects all the useful test predicates and, therefore, it does not impact over the test suite size.

O.13 *Checking uniqueness after N*

This optimization starts checking the uniqueness of the model only after N test predicates are added to the collection. The first N predicates are added (if possible) in the classic way. After that, the witness of the collection is checked for its uniqueness: if it is unique, the collecting stops.

14.4 Experimental results

To evaluate the approach described in this work, we perform a set of experiments, trying to measure the effects of the described optimizations (except O.5 which is always applied).

We first present the set of benchmarks we consider and the SAT and SMT solvers selected. In Section 14.4.1, we describe some basic results about those trivial optimizations that, when possible, must be applied. In Section 14.4.2, we compare the different solvers, evaluating the efficiency of optimizations O.3 and O.4: we first compare all the solvers and then we compare those three that, in the first step, performed better. In Section 14.4.3, we evaluate the optimizations O.1, O.6, O.7, O.9 and O.10, using the solver that had the best performances in experiment of Section 14.4.2. In Section 14.4.4, we evaluate the optimizations O.11, O.12 and O.13 regarding the limiting of the collecting process. Finally, in Section 14.4.5, we evaluate how good is our collecting algorithm compared to the minimal solution.

We run all the experiments on a Linux PC with 8 Intel(R) Xeon(R) CPUs E5430 @ 2.66GHz and 8 GB of RAM.

Solver	API	GF (vs CNF)	incremental	backtracking
Optimization	O.4	O.3	O.8	O.9
SAT4J	Y	N	Y	N
MiniSAT	Y/N	N	Y	N
NFLSAT	N	Y	N	N
PicoSAT	Y/N	N	Y	N
Yices	Y/N	Y	Y	Y

Table 14.1: Solver features

Boolean expressions For experimentation, we consider the same set of predicates introduced by [167], who selected 13 Boolean conditions from the specification of TCAS II, an aircraft collision avoidance system. They also added 7 specifications after having identified variable dependencies. This set was originally used by [167] to evaluate several testing criteria and generation techniques, and the same set is still commonly used as benchmark for test generation techniques. Although SAT and SMT solvers can deal with much more complex Boolean expressions than those of TCAS, we decide to consider the latter as benchmark since we believe the TCAS specifications are a meaningful sample of the complexity of Boolean conditions usually occurring in actual software. In any case, besides the TCAS specifications, we also consider Boolean expressions randomly generated. The complexity of these random expressions grows increasing the *number of variables* occurring in the expression and the expression *depth*, namely the height of the expression syntactic tree.

SAT/SMT solvers As SAT solvers we select SAT4J [121], MiniSAT [63], PicoSAT [28] and NFLSAT [101]. SAT4J [121] is a mature, open source library of SAT-based solvers for Java, SAT4J can be embedded in Java and does not require any exchange of file. However, it requires that the test predicates are transformed into CNF. MiniSAT is a minimalistic, open-source SAT solver, which proved to be very effective in all the SAT competitions over the past years. We add also PicoSAT since it claims to support incremental SAT checking. As SMT solver, we decide to use Yices [60], which we have also used in the past and which includes a very efficient SAT solver; it claims to be *competitive as an ordinary SAT and MaxSAT solver* [60]. To implement optimization O.4, we use the solvers (if possible) together with the Java Native Access (JNA) libraries, which simply require native shared libraries.

A brief comparison of the capabilities of the solvers is reported in Table 14.1; Y/N in the column API means that the solver can be used both with the command line version, or accessed through its APIs. MiniSAT and PicoSAT support a limited form of backtracking when all the added constraints are unit clauses and this is not enough for implementing O.8. SAT4J incremental does not work as expected. NFLSAT does not require inputs as CNF, but it cannot work with JNA since it comes as executable binary. Yices has a very rich API, accepts GF Boolean expressions and supports a very efficient backtracking technique.

14.4.1 Basic optimizations

First of all, we want to assess the influence of O.1 and of the transformations to CNF (O.2). We test two CNF transformations, one that preserves equivalence and the Tseitin transformation. The experiment reveals that *O.1 always speeds up the generation process* and that the test generation made with *the Tseitin algorithm is faster* (by around 75%) than the generation made with a normal CNF translation algorithm, even if it increases the number of literals.

For the rest of reported experimentations, we assume the use of O.1 and of the Tseitin transformation to CNF (if applicable), unless stated otherwise.

14.4.2 Comparing the solvers

The second set of experiments aims at comparing all the solvers using only optimizations O.3 and O.4 whenever possible. The goal is to provide evidence of the effectiveness of the two options and to identify the best solver.

Regarding O.3, Yices and NFLSAT accept Boolean predicates in general form, while the others require inputs in CNF. We apply O.4 to Yices, MiniSAT, and PicoSAT by using two versions: one at command line (CLI) and the other one using JNA. SAT4J can be used only as Java library, while NFLSAT can be used only at command line.

For the 8 possible solver configurations, Fig. 14.2 reports the time taken to generate the complete test suite for the 20 TCAS specifications for 50 runs with a timeout of 1 hour for every run. NFLSAT completes all the specifications without any timeout with an average time of 189 mins; MiniSAT and PicoSAT in the command line version, instead, run out of time for 2 specifications and they take, respectively, an average time of 259 mins and 280 mins. Yices in CLI complete all the specifications without any timeout with an average time of 215 mins.

Optimization O.3 has a positive effect: NFLSAT and Yices, which take GF predicates, perform better than the other solvers that accept only CNF. The SAT solver NFLAST performs even better than the SMT solver Yices.

However, NFLSAT is much slower than the other solvers when applying O.4 which is not supported by NFLSAT. Unfortunately NFLSAT is closed source, but it would be interesting to experiment a library version of NFLSAT.

All the solvers with JNA perform considerably better than the CLI counterparts: *O.4 drastically reduces the time necessary to generate the tests.* With O.4, only PicoSAT runs out of time for the TCAS specifications.

In order to thoroughly compare the best solvers (SAT4J, Yices and MiniSAT with JNA), we perform another experiment. We select 12200 random specifications in the following way: we identify 610 combinations ($\#ids$, $depth$) where $\#ids$ is the number of variables in the formula. To avoid having complex expressions with few variables, we impose the constraints: $1 \leqslant \#ids \leqslant 20$ and $1 \leqslant depth \leqslant min(50, 7 \cdot \#ids)$. For each combination ($\#ids$, $depth$), we select 20 specifications.

Fig. 14.3 reports the total time for the test generation. Yices performs better than the other two SAT solvers: *SMT solving is competitive even for test generation for Boolean expressions.*

In conclusion, experiments reveal that probably what is lost by using an SMT solver, is gained by avoiding the conversion to CNF. We, therefore, choose Yices to perform the remaining experiments carried out to compare more deeply the optimizations regarding the collecting algorithm.

14.4.3 Optimization evaluation

We run the test generation algorithm over 40 specifications (the 20 TCAS specifications and those 20 random specifications that, in the experiment described in Section 14.4.2, were the toughest to solve) for 50 runs applying the optimizations O.1, O.6, O.7, and incremental collecting in three variants: not applied, single incremental collecting (O.9), and double incremental collecting (O.10). Totally, there are 24 possible combinations of optimizations, but only 14 are feasible: indeed, the optimization O.1 cannot be applied together with the double collecting O.10 because double collecting requires the test predicates in the *xor* form, and optimization O.7 cannot be applied with incremental collecting because a test predicate must be added in any case to the logical context in order to allow incremental construction. We do not experiment O.8 since it is superseded by O.9 and because it has proved to be ineffective in experiments not reported here. Fig. 14.4 reports the time required to complete the test generation for all the 40 specifications depending on the optimizations used. The grey box enlarges the cases in which incremental collecting is applied. It is evident that the incremental collecting significantly improves the performances.

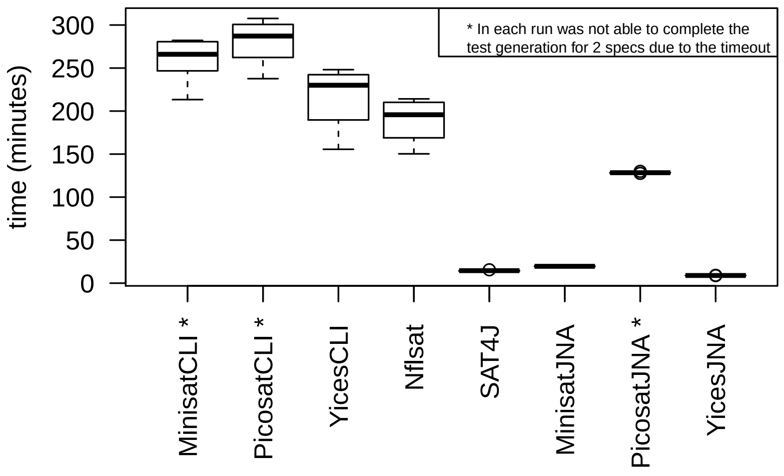


Figure 14.2: Comparing the solvers with TCAS

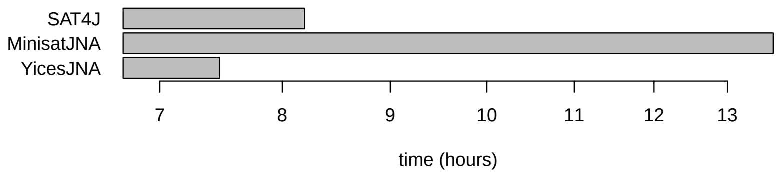


Figure 14.3: Comparing the best solvers with random specifications

Table 14.2 reports the observed improvements due to every single optimization O.*i*. It compares the time (as average and deviation) required to complete the test generation for all the specifications when O.*i* is not applied (column *without*) with the time when O.*i* is applied (column *with*), considering all the combinations of the other optimizations where O.*i* is applicable.

The only ineffective optimization is O.6: it seems that an extra call of the SMT solver after the collecting phase does not impact over the final time.

The other options are effective. Checking if the witness of the collection is also a valid model for the test predicate being added to the collection (O.7) is the least effective. The simplification of the *xor* operator (O.1) significantly improves the performance, as we already experimented using all the solvers in the first experiment set. This optimization could be directly embedded in the SMT solver, which probably does not apply this form of simplification because it is rarely applicable for generic Boolean expressions. The *incremental collecting (O.9 and O.10) boosts the performances* by significantly reducing the time. It seems that double incremental collecting is slightly more efficient than single incremental collecting.

Optimization	Avg. time (min:sec)		Improvement	
	without	with	avg.	dev.
O.1: *xor* simplification	16:54	11:04	34.5%	0.64%
O.6: Collection with witness	12:06	12:10	-0.4%	0.96%
O.7: Checking if the witness is a model	22:12	18:34	16.3%	0.76%
O.9: Single incremental collecting	20:23	1:11	94.2%	0.15%
O.10: Double incremental collecting	20:23	1:02	94.9%	0.14%

Table 14.2: Optimization improvements

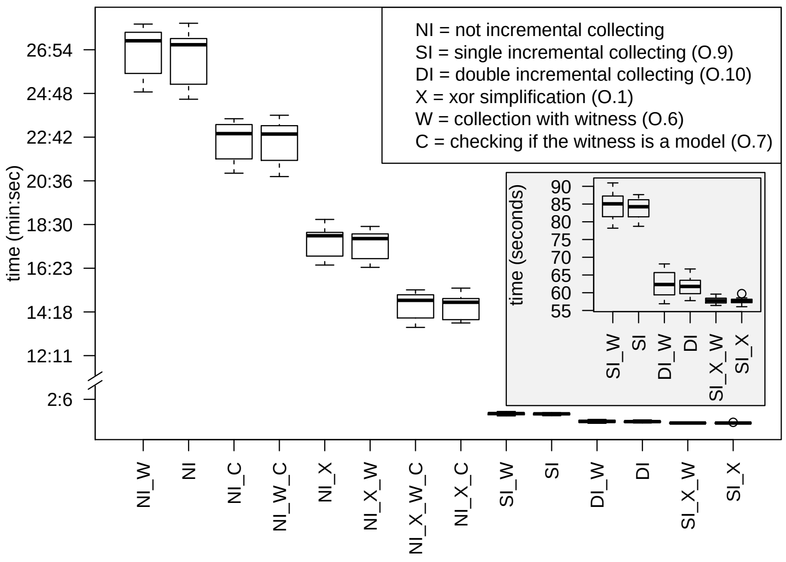


Figure 14.4: Optimization evaluation with Yices

Optimizations		Time (secs)		
Collecting	others	average	min	deviation
Single (O.9)	O.1	57.6	56.0	± 0.82
Double (O.10)		61.9	57.7	± 2.57

Table 14.3: Time for best optimizations

However, double collecting cannot be used in conjunction with the *xor* simplification (O.1) and this has the effect that *the best performances are obtained when the single collecting (O.9) is applied together with O.1* (as reported in Table 14.3). In this case, the generation for all the test suites for all the 40 specifications requires only 57.6 secs, with an average of 1.44 secs for specification. This proofs that a well engineered and optimized SMT-based test generation process can be used in practice for Boolean specifications instead of the classical algorithms.

14.4.4 Limiting collecting

In this experiment, we test how all the limiting policies introduced in Section 14.3.2 may affect the test generation process (used together with O.9). We run the test generation for all the 40 specifications for 50 runs with O.11, O.12, and O.13 with N from 1 to 60. Table 14.4 reports the test suite size and the time (average and deviation). These policies are compared (column ± wrt §) with the best results obtained by single incremental collecting (reported again in the first row § in the table).

Fig. 14.5 depicts the effects of O.11. As the figure shows and as expected, the size of the test suite decreases with increasing N, but the time required increases as well. For small N, the size rapidly decreases, but after a threshold (around 15) the test suite size is reduced only marginally. This option gives the user more control over the collecting process: a suitable value of N can be chosen to balance between test suite compactness and test generation time. Note that the test suite becomes of comparable size w.r.t. the case without limiting when N approaches 60, but the generation time still remains smaller.

O.12 can find a test suite as small as the best combination §, but the time required is almost tripled.

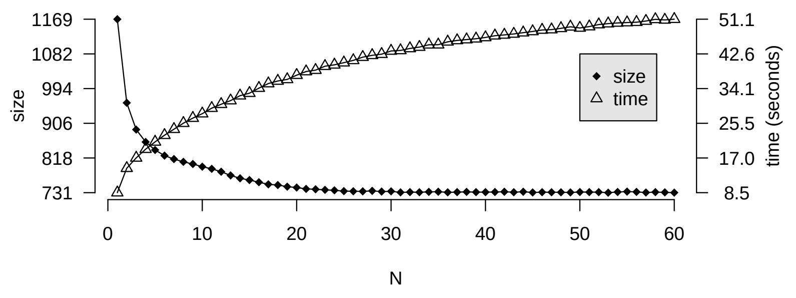


Figure 14.5: Quit collecting after N (O.11)

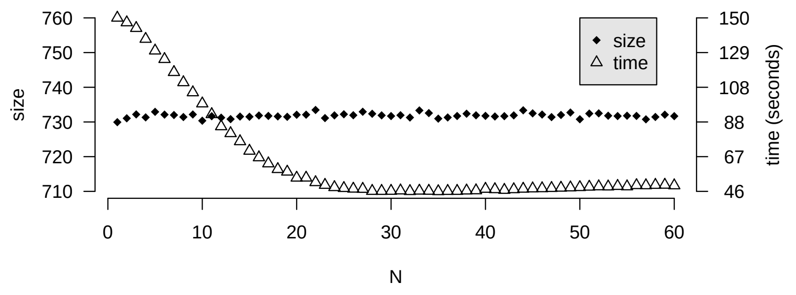


Figure 14.6: Checking uniqueness after N (O.13)

Fig. 14.6 depicts the effects of O.13 which is capable of finding test suites as small as the best combination §; for values of N lower than around 20 the time is greater, for values of N greater than 20 the time is lower. We can argue that the collections contain, on average, 20 test predicates. Indeed, for values of N lower than 20 the times for O.13 are greater than the time of the best combination §: this means that, most of the times, the uniqueness check fails (the model is not unique) and the time taken by the check is wasted. For values of N greater than 20, instead, the times for O.13 are lower than the time of the best combination §: this means that, most of the times, the uniqueness check succeed (the model is unique) and the collecting can be interrupted, while in the best combination § the collecting must be executed on all the test predicates.

Overall, we can state that *limiting the collecting is effective in reducing the time for test generation* with possible no negative effects over the test suite size.

14.4.5 Optimality of the collecting process

As mentioned in Section 14.1.1, the optimal partition of the test predicate set would guarantee the minimal number of partitions and this would ensure the minimality of the final test suite. The proposed greedy collecting process depends on the order in which test predicates are considered and it may not find the optimal partition of the test predicates. We are interested in measuring how much the solutions computed by such greedy algorithm differ from the optimal ones.

With this goal, we run the test generation process 2000 times for all the 40 specifications. For each specification we identify, among the 2000 runs, the smallest value for the test suite size which is likely very close or equal to the minimal optimal value. Then we measure, in all the runs, the distances between the obtained values and the minimum. In Fig. 14.7 we show the cumulative distribution of the difference of the test suite size for every run and for every specification w.r.t. its minimum. As shown by Fig. 14.7, for the 43% of the cases we obtain the minimum, for the 90% of the cases the size of the test suite is maximum 20% bigger than the (optimal) smallest test suite. In particular cases, it may be necessary to use some heuristics in test predicate ordering to guarantee better results.

Policy	Test suite Size			Time (secs)		
	average	dev.	± wrt§	average	dev.	± wrt§
§ no limits (O.9)	731.45	5.37		57.65	0.82	
Fixed N (O.11)						
1	1169.42	5.16	59.88%	8.53	0.35	-85.21%
3	890.18	6.75	21.7%	17.06	0.43	-70.41%
7	815.6	5.79	11.5%	24.1	0.54	-58.2%
10	796.34	6.53	8.87%	27.91	0.75	-51.58%
15	762.54	5.16	4.25%	32.94	0.77	-42.87%
20	743.56	6.42	1.66%	37.35	0.88	-35.2%
30	734	5.2	0.35%	43.29	1.1	-24.91%
50	732.18	5.34	0.1%	48.93	1.06	-15.12%
60	730.64	5.18	-0.11%	51.08	1.31	-11.39%
Until unique (O.12)	731.54	4.52	0.01%	158.2	7.36	174.42%
Mixed (O.13)						
1	729.94	4.27	-0.21%	150.11	6.31	160.38%
3	732.18	4.37	0.1%	143.93	6.89	149.68%
7	732	5.18	0.08%	117.47	6.02	103.77%
10	730.36	4.54	-0.15%	98.64	7.7	71.11%
15	731.5	5.21	0.01%	70.22	5.9	21.81%
20	732.08	5.38	0.09%	54.08	3.91	-6.2%
30	731.7	4.45	0.03%	46.15	2.13	-19.94%
50	730.78	4.59	-0.09%	48.41	1.39	-16.02%
60	731.68	4.33	0.03%	49.45	1.17	-14.21%

Table 14.4: Comparison of limiting policies

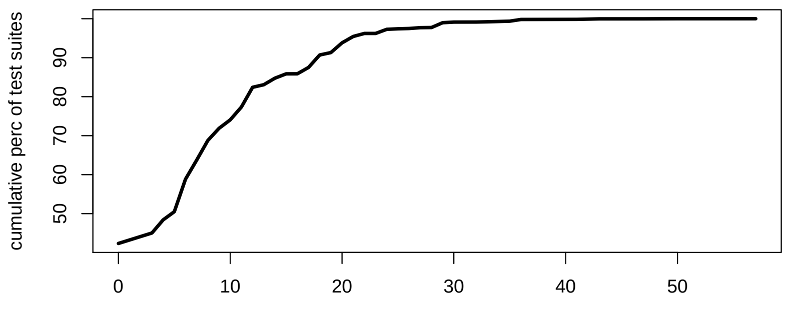


Figure 14.7: Optimality of the Collecting Process

Conclusions

The works presented in this thesis have been published in different forms in journal papers, conference papers, technical reports, and book chapters. The list of publications is reported in Appendix A.

In the following, we recap the contribution of the thesis and describe some possible future work.

Contribution of the thesis

With the aim of achieving the principal four goals presented in Section 1.2, the current thesis has proposed the following contributions.

1. A process, based on Model-Driven Engineering, for the development of a toolset around a formal method. The proposed process permits to obtain (in a (semi)-automatic way) several tools, which support different activities of a system development process, from system specification to system analysis; it also fosters software reuse, since several software artifacts are reused by all the tools. The proposed approach has been instantiated for ASMs and FSMs, but it can be used for any FM.

2. A model checker for ASMs. This works fosters the integration between formal methods, by providing a mapping between AsmetaL models and NuSMV models, so that the model checking can be executed directly on the high-level notation provided by ASMs.

3. A model review approach for state-based formal methods to check that a model of a FM has some *quality* attributes that any model of that FM should have. The general technique has been instantiated for NuSMV models and ASMs. These works address the goal of helping the user in writing correct specifications.

   (a) The approach has been evaluated with a new technique that we propose to assess the fault detection capability of a static analysis technique using mutation analysis.

4. A runtime monitoring technique which uses ASMs as specification language of the expected behaviour. This work addresses the goal of reducing the distance between the formal specification and the actual implementation of the system, by providing a mechanism to link them, and an approach for checking if they are conformant.

   (a) The approach has been combined with Model-Based Testing to test nondeterministic Java programs.

Moreover, two approaches for addressing scalability issues in formal analysis techniques have been proposed: an approach to mitigate the state space explosion problem in the contest of test generation from ASMs using model checkers, and some optimizations for the process of test generation for boolean expressions using SAT/SMT solvers.

Future work

We here describe some of the possible improvements of the works presented in this thesis.

Model review The model review techniques could be extended by adding new meta-properties addressing particular errors that a user would like to look for. In the ASMs, for example, one of the typical shortcomings introduced by a not ASM-expert is the use of the sequential rule when the block (parallel) rule could be used instead. This is usually due to a wrong understanding of the simultaneous parallel execution of function updates. The correct use of a parallel rule, instead of a sequential rule, can improve the quality of a model in terms of abstraction and minimality. So we plan to investigate this kind of defect and define suitable meta-properties able to detect it.

Assessing fault detection capability using mutation analysis In the proposed approach we have used only first order mutants, i.e., mutants obtained from the injection of a single fault in the original specification. We plan to experiment our methodology using higher order mutants, i.e., mutants obtained by the injection of more than a fault. Since the number of obtained mutants could be very high, we should define some heuristics to select only some of them. Moreover we should try to avoid checking the equivalence for all the considered mutants, establishing some relations between first order and higher order mutants: the idea is that, if a first order mutant is not equivalent, we should be able to establish that some of the higher order mutants obtained from it are not equivalent as well.

The presented approach has furnished us a lot of statistical data that relate the meta-properties with the considered faults. Starting from these data, we plan to define a procedure that, given some meta-properties violations, provides the user with an estimate of the kind of faults occurring in the specification, and possibly advises her on how to correct them. This procedure could improve the quality of the results returned by the model advisor.

Model checking ASMs The AsmetaSMV tool can currently translate only a limited set of ASMs into NuSMV. In particular it does not permit to translate turbo rules, except a limited support of the sequential rule. In order to handle turbo rules, rather than trying to map them directly into NuSMV, we could implement a mechanism that maps ASMs containing turbo rules into equivalent ASMs containing only basic rules: these kind of ASMs is fully supported by the tool.

The NuSMV models produced by the tool are often not minimal, i.e., they contain elements that have been introduced by the mapping process but are never used (e.g., some values of a variable type). In order to achieve minimality of the obtained models, we plan to integrate AsmetaSMV with the NuSMV model advisor, so that the results of the model review over the NuSMV model can be used to identify unnecessary elements and remove them.

Runtime monitoring The proposed runtime monitoring approach has some limits. Since each class is linked to its specification, monitoring safety properties involving collections of two or more objects [46] is not possible, but we plan to extend CoMA to support also these scenarios. Moreover, monitoring real time requirements seems problematic: we believe that a monitored function *time* may model the real time and would allow its measurement, but further experiences are needed. Finally, the main limitation of CoMA is that, since it currently checks conformance by interpreting the ASM, it performs much slower than other approaches. We plan to optimize the monitoring process to reduce the temporal overhead by, for example, encoding the ASM directly in Java.

Currently CoMA, in case of nondeterministic specification, only supports strong conformance, i.e., it requires that it exists only one next state of the ASM that is conformant with the Java state. Weak conformance, requiring that it exists at least one next conformant state, is currently not supported. An extension of CoMA could support weak conformance; to do this we should, when more than one next state is conformant, create different simulations of the ASM, one for each next conformant state. However, this approach would introduce too much time execution overhead when a lot of weak conformant steps are executed.

Bibliography

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 345–364, New York, NY, USA, 2005. ACM.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[3] Android website. `http://www.android.com/`.

[4] The Apache Axis2 website. `http://axis.apache.org/axis2/java/core/`, 2012.

[5] P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a model checker for AsmetaL models. tutorial. TR 120, Dipartimento di Tecnologie dell'Informazione, Università degli Studi di Milano, 2009.

[6] P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2010.

[7] P. Arcaini, A. Gargantini, and E. Riccobene. Automatic Review of Abstract State Machines by Meta Property Verification. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 4–13, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[8] P. Arcaini, A. Gargantini, and E. Riccobene. Runtime monitoring of Java programs by Abstract State Machines. TR 131, Dipartimento di Tecnologie dell'Informazione, Università degli Studi di Milano, 2010.

[9] P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41(2):155–166, 2011.

[10] P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. Formal semantics for metamodel-based domain specific languages. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments.* IGI Global, 2012.

[11] Arvind, N. Dave, and M. Katelman. Getting Formal Verification into Design Flow. In *Proceedings of the 15th international symposium on Formal Methods*, FM '08, pages 12–32, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] AsmetaL documentation. `http://asmeta.sourceforge.net/userdoc/language.html`, 2012.

[13] The ASMETA website. `http://asmeta.sourceforge.net/`, 2012.

[14] The ASML language. `http://research.microsoft.com/en-us/projects/asml/`, 2001.

[15] A. Aziz, V. Singhal, and F. Balarin. Equivalences for fair kripke structures. In *International Colloquium on Automata, Languages and Programming*, pages 364–375. Springer-Verlag, 1994.

[16] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[17] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Software*, 1(6):219–232, 2007.

[18] L. Baresi and S. Guinea. Dynamo: Dynamic Monitoring of WS-BPEL Processes. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC'05)*, pages 478–483. Springer, 2005.

[19] L. Baresi and S. Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. In B. Benatallah, F. Casati, and P. Traverso, editors, *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282. Springer Berlin / Heidelberg, 2005.

[20] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, Mar. 2003.

[21] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. Technical Report TUM-I0724, Institut für Informatik, Technische Universität München, Dec. 2007.

[22] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software and Methodology (TOSEM)*, 20, 2011.

[23] J. Beckers, D. Klünder, S. Kowalewski, and B. Schlich. Direct support for model checking Abstract State Machines by utilizing simulation. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, volume 5238 of *LNCS*, pages 112–124. Springer, 2008.

[24] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient Detection of Vacuity in ACTL Formulaas. In O. Grümberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 279–290. Springer, 1997.

[25] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Proceedings 9th International Computer Aided Verification Conference*, number 1254 in Lecture Notes in Computer Science, pages 279–290, 1997.

[26] T. Berg and H. Raffelt. Model checking. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 557–603. Springer, 2004.

[27] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with bdds for automatic invariant checking. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, TACAS '00, pages 378–394, London, UK, UK, 2000. Springer-Verlag.

[28] A. Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[29] B. W. Boehm. Software Engineering; R & D trends and defense needs. In P. Wegner, editor, *Research Directions in Software Technology (Ch. 22)*, pages 1–9, Cambridge, MA, 1979. MIT Press.

[30] E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2005.

[31] E. Börger. The Abstract State Machines Method for High-Level System Design and Analysis. Technical report, BCS Facs Seminar Series Book, 2007.

[32] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer Verlag, 2003.

[33] J. P. Bowen and M. G. Hinchey. Ten Commandments of Formal Methods. *Computer*, 28(4):56–63, Apr. 1995.

[34] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods ...ten years later. *Computer*, 39(1):40–48, Jan. 2006.

[35] M. Brörkens and M. Möller. Dynamic event generation for runtime checking using the JDI. *Electronic Notes in Theoretical Computer Science*, 70(4):21–35, 2002.

[36] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1-2):115 – 131, 1988.

[37] T. Bultan and C. Heitmeyer. Applying infinite state model checking and other analysis techniques to tabular requirements specifications of safety-critical systems. *Design Automation for Embedded Systems*, 12(1-2):97–137, 2008.

[38] A. Calvagna and A. Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, 2010.

[39] A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A scenario-based validation language for ASMs. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, volume 5238 of *LNCS*, pages 71–84. Springer, 2008.

[40] G. D. Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000*, volume 1785 of *LNCS*, pages 331–346. Springer, 2000.

[41] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. NuSMV 2.5 User Manual. `http://nusmv.fbk.eu/`, 2012.

[42] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, and M. Roveri. NuSMV 2.5 Tutorial. `http://nusmv.fbk.eu/`, 2012.

[43] F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 357–372. Springer Berlin / Heidelberg, 2004.

[44] F. Chen, D. Jin, P. Meredith, and G. Roşu. Monitoring oriented programming - a project overview. In *Proceedings of the Fourth International Conference on Intelligent Computing and Information Systems (ICICIS'09)*, pages 72–77. ACM, 2009.

[45] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 2005.

[46] F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07*, page 569, Montreal, Quebec, Canada, 2007.

[47] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3), 1979.

[48] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*. Springer, July 2002.

[49] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[50] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2008.

[51] E. M. Clarke, O. Grümberg, and D. Peled. *Model checking.* MIT Press, 2001.

[52] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 129–139, New York, NY, USA, 2007. ACM.

[53] S. Colin and L. Mariani. Run-time verification. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer Berlin / Heidelberg, 2005.

[54] A. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pages 336–351, Berlin, Heidelberg, 2007. Springer-Verlag.

[55] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[56] S. D. R. S. De Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. L. De Souza. Mutation Testing Applied to Estelle Specifications. *Software Quality Control*, 8:285–301, December 1999.

[57] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.

[58] S. Dewhurst. *C++ Gotchas: Avoiding Common Problems in Coding and Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[59] G. A. Di Lucca and A. R. Fasolino. Testing Web-based applications: The state of the art and future trends. *Information and Software Technology*, 48:1172–1186, 2006.

[60] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI Available at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[61] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[62] The Eclemma website. `http://www.eclemma.org/`, 2012.

[63] N. Een and N. Sörensson. Minisat v2. 0 (beta). *Solver description, SAT race*, 2006.

[64] S. C. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero. Mutation analysis testing for finite state machines. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 220 –229, nov 1994.

[65] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and E. Wong. Mutation Testing Applied to Validate Specifications Based on Petri Nets. In *Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII*, pages 329–337, London, UK, 1996. Chapman & Hall, Ltd.

[66] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Proceedings 10th Int Software Reliability Engineering Symp*, pages 210–219, 1999.

[67] M. E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, July 1986.

[68] R. Farahbod and U. Glässer. The CoreASM modeling framework. *Software: Practice and Experience*, 41(2):167–178, 2011.

[69] R. Farahbod, U. Glässer, and G. Ma. Model Checking CoreASM Specifications. In A. Prinz, editor, *Proceedings of the ASM'07, The 14th International ASM Workshop*, 2007.

[70] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of ACM SIGSOFT ESEC/FSE*, page 416–419, 2011.

[71] G. Fraser and A. Gargantini. An evaluation of model checkers for specification based test case generation. In *Proceedings of the Second International Conference on Software Testing Verification and Validation (ICST 2009), Denver, Colorado, USA*, pages 41–50. IEEE Computer Society, 2009.

[72] G. Fraser and A. Gargantini. Generating minimal fault detecting test suites for boolean expressions. In *6th Workshop on Advances in Model Based Testing A-MOST 2010*. IEEE Computer Society, 2010.

[73] G. Fraser and F. Wotawa. Nondeterministic testing with linear model-checker counterexamples. In *Proceedings of the 7th International Conference on Quality Software*, QSIC '07, pages 107–116, Washington, DC, USA, 2007. IEEE Computer Society.

[74] G. Fraser and F. Wotawa. Test-case generation and coverage analysis for nondeterministic systems using model-checkers. In *Proceedings of the International Conference on Software Engineering Advances*, ICSEA '07, pages 45–, Washington, DC, USA, 2007. IEEE Computer Society.

[75] A. Gargantini and C. L. Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Proceedings of Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly*

with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 1999.

[76] A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal for Universal Computer Science (J.UCS)*, 7:262–265, 2001.

[77] A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to generate tests from ASM specifications. In *Abstract State Machines, Advances in Theory and Practice*, number 2589 in LNCS, pages 263–277. Springer, 2003.

[78] A. Gargantini, E. Riccobene, and P. Scandurra. Deriving a textual notation from a metamodel: an experience on bridging Modelware and Grammarware. In *3M4MDA'06 workshop at the European Conference on MDA*, 2006.

[79] A. Gargantini, E. Riccobene, and P. Scandurra. Metamodelling a Formal Method: Applying MDE to Abstract State Machines. Technical Report 97, Dipartimento di Tecnologie dell'Informazione, Università degli Studi di Milano, 2006.

[80] A. Gargantini, E. Riccobene, and P. Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *Journal for Universal Computer Science (J.UCS)*, 14(12):1949–1983, 2008.

[81] A. Gargantini, E. Riccobene, and P. Scandurra. Model-driven language engineering: The ASMETA case study. In *International Conference on Software Engineering Advances, ICSEA*, pages 373–378, 2008.

[82] M. Gheorghiu and A. Gurfinkel. VaqUoT: A Tool for Vacuity Detection. In *Posters & Research Tools Track, FM 2006*, 2006.

[83] C. Ghezzi and S. Guinea. Run-time monitoring in service-oriented architectures. In L. Baresi and E. Di Nitto, editors, *Test and Analysis of Web Services*, pages 237–264. Springer, 2007.

[84] U. Glässer, Y. Gurevich, and M. Veanes. High-level executable specification of the universal plug and play architecture. In *HICSS*, page 283, 2002.

[85] B. J. Gruen, D. Schuler, and A. Zeller. The Impact of Equivalent Mutants. In *Mutation '09: Proceedings of the 3rd International Workshop on Mutation Analysis*, pages 192–199, April 2009.

[86] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[87] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1):77–111, 2000.

[88] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[89] S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire. Runtime Verification of Web Service Interface Contracts. *Computer*, 43(3):59 –66, 2010.

[90] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[91] M. J. Harrold, R. Gupta, and M. L. Soffa. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

[92] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6:158–173, August 2004.

[93] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *ECMDA-FA*, 2009.

[94] C. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

[95] C. L. Heitmeyer. On the need for practical formal methods. In *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, FTRTFT '98, pages 18–26, London, UK, 1998. Springer-Verlag.

[96] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):9:1–9:76, Feb. 2009.

[97] T. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall International, Englewood Cliffs, NJ, 1998.

[98] T. Hoare and J. Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, 2005*, volume 4171 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[99] G. Holzmann. *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional, first edition, 2003.

[100] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, New York, NY, USA, 2004.

[101] H. Jain and E. M. Clarke. Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts. In *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC '09)*, pages 563–568, 2009.

[102] Java Annotations tutorial. `http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html`, 2012.

[103] The JavaMOP website. `http://fsl.cs.uiuc.edu/index.php/Special:JavaMOP3`, 2012.

[104] JCGM 200:2008 International vocabulary of metrology. Basic and general concepts and associated terms (VIM).

[105] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379 – 1393, 2009.

[106] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37:649–678, 2011.

[107] F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the fifth international conference on Generative programming and Component Engineering (GPCE'06)*, 2006.

[108] K. Kähkönen, J. Lampinen, K. Heljanko, and I. Niemelä. The LIME interface specification language and runtime monitoring tool. In S. Bensalem and D. A. Peled, editors, *Runtime Verification*, volume 5779, chapter 7, pages 93–100. Springer, Berlin, Heidelberg, 2009.

[109] G. K. Kaminski and P. Ammann. Using logic criterion feasibility to reduce test set size while guaranteeing fault detection. In *ICST*, pages 356–365. IEEE Computer Society, Apr. 1–4, 2009.

[110] K. Kapoor and J. P. Bowen. Test conditions for fault classes in Boolean specifications. *ACM Transactions on Software Engineering and Methodology*, 16(3):10, 2007.

[111] M. Kardos. An approach to model checking asml specifications. In *Abstract State Machines*, pages 289–304, 2005.

[112] M. Kaufmann and J. S. Moore. Some key research problems in automated theorem proving for hardware and software verification. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales. Serie A: Matemáticas (RACSAM)*, 98(1):181–195, 2004.

[113] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, 1997.

[114] T. Kim and S. D. Cha. Automated structural analysis of SCR-style software requirements specifications using PVS. *Software Testing, Verification and Reliability*, 11(3):143–163, 2001.

[115] J. C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA, 2002. ACM.

[116] W. H. Kruskal and W. A. Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, pages 583–621, 1952.

[117] D. R. Kuhn. Fault Classes and Error Detection Capability of Specification-Based Testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, 1999.

[118] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, 2003.

[119] K. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin / Heidelberg, 2005.

[120] M. F. Lau and Y.-T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 14(3):247–276, 2005.

[121] D. Le Berre and A. Parrain. The SAT4J library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation(JSAT)*, 7:59–64, 2010.

[122] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31:1–38, May 2006.

[123] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1999, Las Vegas, Nevada, USA*, pages 279–287. CSREA Press, 1999.

[124] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.

[125] M. Leuschel. The High Road to Formal Validation. In *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, ABZ '08, pages 4–23, Berlin, Heidelberg, 2008. Springer-Verlag.

[126] H. Liang, J. Dong, J. Sun, and W. Wong. Software monitoring through formal specification animation. *Innovations in Systems and Software Engineering*, 5:231–241, 2009.

[127] L. Liu and H. Miao. Mutation operators for Object-Z specification. In *Proceedings 10th IEEE International Conference on Engineering of Complex Computer Systems ICECCS 2005*, pages 498–506, 2005.

[128] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[129] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[130] A. M. Memon and O. Akinmade. Automated Model-Based Testing of Web Applications. In *Google Test Automation Conference 2008*, 2008.

[131] S. Merz. Model checking: A tutorial overview. In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, MOVEP '00, pages 3–38, London, UK, UK, 2001. Springer-Verlag.

[132] R. Miles. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[133] Mastercard international inc.: Mondex. `http://www.mondex.com/`.

[134] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153, Princeton, 1956.

[135] M. Müller-Olm, D. A. Schmidt, and B. Steffen. Model-checking: A tutorial introduction. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, pages 330–354, London, UK, UK, 1999. Springer-Verlag.

[136] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[137] The NuSMV website. `http://nusmv.fbk.eu/`, 2012.

[138] The nusmv-tools website. `http://code.google.com/a/eclipselabs.org/p/nusmv-tools/`, 2012.

[139] OMG. Object Constraint Language (OCL), v2.0 formal/2006-05-01, 2006.

[140] A. J. Offutt. The coupling effect: fact or fiction. *SIGSOFT Software Engineering Notes*, 14:131–140, November 1989.

[141] M. Ouimet and K. Lundqvist. Formal software verification: Model checking and theorem proving. Technical Report, Målardalen University, March 2007.

[142] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE-11)*, pages 748–752, London, UK, 1992. Springer-Verlag.

[143] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[144] D. L. Parnas. Some Theorems We Should Prove. In *HUG '93: 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 155–162, London, UK, 1994. Springer-Verlag.

[145] D. L. Parnas. "Formal methods" technology transfer will fail. *Journal of Systems and Software*, 40(3):195 – 198, 1998.

[146] D. A. Peled. *Software reliability methods.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[147] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Roşu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS '08, pages 481–491, Washington, DC, USA, 2008. IEEE Computer Society.

[148] S. Perera, C. Herath, J. Ekanayake, E. Chinthaka, A. Ranabahu, D. Jayasinghe, S. Weerawarana, and G. Daniels. Axis2, middleware for next generation web services. In *Proceedings of the IEEE International Conference on Web Services*, pages 833–840, Washington, DC, USA, 2006. IEEE Computer Society.

[149] A. Pnueli. The temporal logic of programs. In *FOCS, 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA*, pages 46–57. IEEE Computer Society, 1977.

[150] S. Prestwich. *Handbook of satisfiability*, chapter 2 - CNF Encodings, pages 75–98. IOS Press, 2009.

[151] S. Prochnow, G. Schaefer, K. Bell, and R. von Hanxleden. Analyzing robustness of UML State Machines. In *Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES 06)*, pages 61–80, 2006.

[152] Sahi website. `http://sahi.co.in/`.

[153] Sat competition. `http://www.satcompetition.org/`.

[154] G. Schellhorn and R. Banach. A Concept-Driven Construction of the Mondex Protocol Using Three Refinements. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, volume 5238, pages 57–70, Berlin, 2008. Springer.

[155] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proceedings of ESEC/ ACM FSE '09*, pages 297–298, August 2009.

[156] Smt competition. `http://www.smtcomp.org/`.

[157] I. Sommerville. *Software Engineering (9th Edition).* Addison Wesley, 2010.

[158] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, 2001.

[159] M. Strembeck and U. Zdun. An Approach for the Systematic Development of Domain-Specific Languages. *Software: Practice and Experience*, 39(15), October 2009.

[160] C. K. F. Tang and E. Ternovska. Model checking abstract state machines with answer set programming. *Fundam. Inf.*, 77(1-2):105–141, Jan. 2007.

[161] Textual Editing Framework. `http://www2.informatik.hu-berlin.de/sam/meta-tools/tef`.

[162] J. Tretmans. Model based testing with labelled transition systems. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[163] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

[164] OMG. The Unified Modeling Language (UML), v2.1.2. `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF`, 2007.

[165] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. *ACM SIGSOFT Software Engineering Notes*, 30(5):273–282, Sept. 2005.

[166] E. W. Weisstein. Knight's tour. from MathWorld–A Wolfram Web Resource.

[167] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[168] K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, 3(5):689–701, 1997.

[169] K. Winter. Towards a methodology for model checking ASM: Lessons learned from the FLASH case study. In Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines*, volume 1912 of *Lecture Notes in Computer Science*, pages 341–360. Springer, 2000.

[170] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[171] Xtext. `http://www.eclipse.org/Xtext/`, 2012.

Appendix A

Publications

Refereed International Journals Articles

1. **A Model Advisor for NuSMV Specifications**
   (co-authors: A. Gargantini, E. Riccobene)
   in *Innovations in Systems and Software Engineering*, Springer London, vol. 7 (2011): 97-107

   Abstract: Among possible model validation techniques able to identify defects early in the system development, model review aims also at determining if a model is of sufficient quality, where quality is measured as the absence of certain faults. In this paper, we tackle the problem of automatic reviewing NuSMV formal specifications by developing a *model advisor* which helps to assure given model qualities for NuSMV programs. Vulnerabilities and defects a developer can introduce during the modeling activity using NuSMV are expressed as the violation of formal meta-properties. These meta-properties are then mapped to temporal logic formulas, and the NuSMV model-checker itself is used as the engine of our model advisor to notify meta-properties violations, so revealing the absence of some quality attributes of the specification. As a proof of concept, we also report the result of applying this review process to several NuSMV specifications.

2. **A model-driven process for engineering a tool-set for a formal method**
   (co-authors: A. Gargantini, E. Riccobene, P. Scandurra)
   in *Software: Practice and Experience*, John Wiley & Sons, Ltd., vol. 41, n. 2 (2011): 155-166

   Abstract: This paper presents a model-driven software process suitable to develop a set of integrated tools around a formal method. This process exploits concepts and technologies of the Model-Driven Engineering (MDE) approach, like metamodelling and automatic generation of software artifacts from models. We describe the requirements to fulfill and the development steps of this model-driven process. As a proof-of-concepts, we apply it to the Finite State Machines and we report our experience in engineering a metamodel-based language and a toolset for the Abstract State Machine formal method.

Refereed Papers in Proceedings of International Conference and Workshops

1. **Components monitoring through formal specifications**
   (co-authors: A. Gargantini, E. Riccobene)
   in *Proc. of the Seventeenth International Doctoral Symposium on Components and Architecture (WCOP 2012)*, Bertinoro, Italy, June 25, 2012

   Abstract: The paper presents a specification-based approach for runtime monitoring of components in the field of component-based software engineering. The conformance of a component is checked with respect to a formal specification given in terms of Abstract State Machines. The validity of the approach is proved showing how the technique can be used for the monitoring of web services developed using Axis2. The theoretical approach is implemented in a technical framework where Java annotations are used to link the web service with its formal specification, and AspectJ is used to check the conformance runtime.

2. **Test Generation for Sequential Nets of Abstract State Machines**
(co-authors: F. Bolis, A. Gargantini)
in *Proc. of the Third International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2012)*, Pisa, Italy, June 18-21, 2012

   Abstract: Test generation techniques based on model checking suffer from the state space explosion problem. However, for a family of systems that can be easily decomposed in sub-systems, we devise a technique to cope with this problem. To model such systems, we introduce the notion of *sequential net* of Abstract State Machines (ASMs), which represents a system constituted by a set of ASMs such that only one ASM is active at every time. Given a net of ASMs, we first generate a test suite for every ASM in the net, then we combine the tests in order to obtain a test suite for the entire system. We prove that, under some assumptions, the technique preserves coverage of the entire system. We test our approach on a benchmark and we report a web application example for which we are able to generate complete test suites.

3. **Optimizing the Automatic Test Generation by SAT and SMT Solving for Boolean Expressions**
(co-authors: A. Gargantini, E. Riccobene)
in *Proc. of the 26th International Conference on Automated Software Engineering (ASE 2011)*, Lawrence, Kansas, November 6 - 12, 2011

   Abstract: Recent advances in propositional satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers are increasingly rendering SAT and SMT-based automatic test generation an attractive alternative to traditional algorithmic test generation methods. The use of SAT/SMT solvers is particularly appealing when testing Boolean expressions: These tools are able to deal with constraints over the models, generate compact test suites, and they support fault-based test generation methods. However, these solvers normally require more time and greater amount of memory than classical test generation algorithms, limiting their applicability. In this paper we propose several ways to optimize the process of test generation and we compare several SAT/SMT solvers and propositional transformation rules. These optimizations promise to make SAT/SMT-based techniques as efficient as standard methods for testing purposes, especially when dealing with Boolean expressions, as proved by our experiments.

4. **CoMA: Conformance Monitoring of Java programs by Abstract State Machines**
(co-authors: A. Gargantini, E. Riccobene)
in *Proc. of the Second International Conference on Runtime Verification (RV 2011)*, San Francisco, California, September 27 - 30, 2011

   Abstract: We present *CoMA* (Conformance Monitoring by Abstract State Machines), a specification-based approach and its supporting tool for runtime monitoring of Java software. Based on the information obtained from code execution and model simulation, the conformance of the concrete implementation is checked with respect to its formal specification given in terms of Abstract State Machines. At runtime, undesirable behaviours of the implementation, as well as incorrect specifications of the system behaviour are recognized. The technique we propose makes use of Java annotations, which link the concrete implementation to its formal model, without enriching the code with behavioural information contained only in the abstract specification. The approach fosters the separation between implementation and specification, and allows the reuse of specifications for other purposes (formal verification, simulation, model-based testing, etc.).

5. **Automatic review of Abstract State Machines by Meta Property Verification**
   (co-authors: A. Gargantini, E. Riccobene)
   in *Proc. of the Second NASA Formal Methods Symposium (NFM 2010)*, Washington D.C., USA, April 13 - 15, 2010

   Abstract: A model review is a validation technique aimed at determining if a model is of sufficient quality and allows defects to be identified early in the system development, reducing the cost of fixing them. In this paper we propose a technique to perform *automatic* review of Abstract State Machine (ASM) formal specifications. We first detect a family of typical vulnerabilities and defects a developer can introduce during the modeling activity using the ASMs and we express such faults as the violation of meta-properties that guarantee certain quality attributes of the specification. These meta-properties are then mapped to temporal logic formulas and model checked for their violation. As a proof of concept, we also report the result of applying this ASM review process to several specifications.

6. **AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications**
   (co-authors: A. Gargantini, E. Riccobene)
   in *Proc. of the Second International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010)*, Orford, QC, Canada, February 22-25, 2010

   Abstract: This paper presents *AsmetaSMV*, a model checker for Abstract State Machines (ASMs). It has been developed with the aim of enriching the ASMETA (ASM mETAmodeling) toolset – a set of tools for ASMs – with the capabilities of the model checker NuSMV to verify properties of ASM models written in the AsmetaL language. We describe the general architecture of AsmetaSMV and the process of automatically mapping ASM models into NuSMV programs. As a proof of concepts, we report the results of using AsmetaSMV to verify temporal properties of various case studies of different characteristics and complexity.

7. **A model-driven process for engineering a tool set for a formal method**
   (co-authors: A. Carioni, A. Gargantini, E. Riccobene, P. Scandurra)
   in *Workshop on Tool Building in Formal Methods (WS-TBFM 2010)*, Orford, QC, Canada, February 22, 2010

   Abstract: We present a model-based software process suitable to develop a set of tools around a formal method. This process is based on the Model-Driven Engineering (MDE) and exploits several concepts and technologies of the MDE, like metamodelling and automatic generation of software artifacts starting from models. The process is the result of our experience in engineering a metamodel-based language and a toolset for the Abstract State Machine formal method.

Chapters in books

1. **Formal Semantics for Metamodel-Based Domain Specific Languages**
   (co-authors: A. Gargantini, E. Riccobene, P.Scandurra)
   in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments* (Eds. Dr. Marjan Mernik). IGI Global (2012)

Abstract: Domain Specific Languages (DSLs) are often defined in terms of metamodels capturing the abstract syntax of the language. For a complete definition of a DSL, both syntactic and semantic aspects of the language have to be specified. Metamodeling environments support syntactic definition issues, but they do not provide any help in defining the semantics of metamodels, which is usually given in natural language. In this chapter, we present an approach to formally define the semantics of metamodel-based languages. It is based on a translational technique that hooks to the language metamodel its precise and executable semantics expressed in terms of the Abstract State Machine formal method. We also show how different techniques can be used for formal analysis of models (i.e., instance of the language metamodel). We exemplify the use of our approach on a language for Petri nets.

Technical reports

1. **Equivalence checking of NuSMV specifications**
   (co-authors: A. Gargantini, E. Riccobene)
   Technical report of Università degli Studi di Milano, Dipartimento di Tecnologie dell'Informazione, no 134 (November 2011)

2. **Runtime monitoring of Java programs by Abstract State Machines**
   (co-authors: A. Gargantini, E. Riccobene)
   Technical report of Università degli Studi di Milano, Dipartimento di Tecnologie dell'Informazione, no 131 (November 2010)

3. **AsmetaSMV : a model checker for AsmetaL models. Tutorial**
   (co-authors: A. Gargantini, E. Riccobene)
   Technical report of Università degli Studi di Milano, Dipartimento di Tecnologie dell'Informazione, no 120 (July 2009)

Papers under submission to international journals

1. **Using Mutation to Assess Fault Detection Capability of Model Review**
   (co-authors: A. Gargantini, E. Riccobene)

Papers under submission to international conferences

1. **Combining Model-Based Testing and Runtime Monitoring for Dealing with Nondeterminism**
   (co-authors: A. Gargantini, E. Riccobene)