AsmetaSMV: a model checker for AsmetaL models
Tutorial

Paolo Arcaini[1] Angelo Gargantini[2] Elvinia Riccobene[3]

[1] Università degli Studi di Milano, Dipartimento di Tecnologie dell'Informazione - parcaini@gmail.com
[2] Università degli Studi di Bergamo, Dipartimento di Ingegneria dell'Informazione e Modelli Matematici - angelo.gargantini@unibg.it
[3] Università degli Studi di Milano, Dipartimento di Tecnologie dell'Informazione - elvinia.riccobene@unimi.it

Contents

Chapter 1

ASMETA Framework

AsmetaSMV is a *based* tool of the ASMETA toolset [1]; so, before describing AsmetaSMV, we briefly introduce the ASMETA toolset.

1.1 ASMETA toolset

ASMETA toolset has been developed starting with the definition of *AsmM* [2, 3], a metamodel for ASMs. The ASMETA tool set (see Fig. 1.1) includes (among other things) a textual concrete syntax, *AsmetaL* [4, 5], to write ASM models (conforming to the AsmM) in a textual and human-comprehensible form; a text-to-model compiler, *AsmetaLc*, to parse AsmetaL models and check for their consistency w.r.t. the AsmM OCL constraints; a simulator, *AsmetaS* [6], to execute ASM models; the *Avalla* language for scenario-based validation of ASM models, with its supporting tool, the *AsmetaV* validator; the *ATGT* tool [7, 8] that is an ASM-based test case generator based upon the SPIN model checker; a graphical front-end called *ASMEE* (ASM Eclipse Environment) which acts as IDE and it is an eclipse plug-in.

All the above artifacts/tools are classified in: *generated*, *based*, and *integrated*. Generated artifacts/tools are derivatives obtained (semi-)automatically by applying appropriate MOF projections to the technical spaces Javaware, XMLware, and grammarware. Based artifacts/tools are those developed exploiting the ASMETA environment and related derivatives; an example of such a tool is the simulator AsmetaS). Integrated artifacts/tools are external and existing tools that are connected to the ASMETA environment.

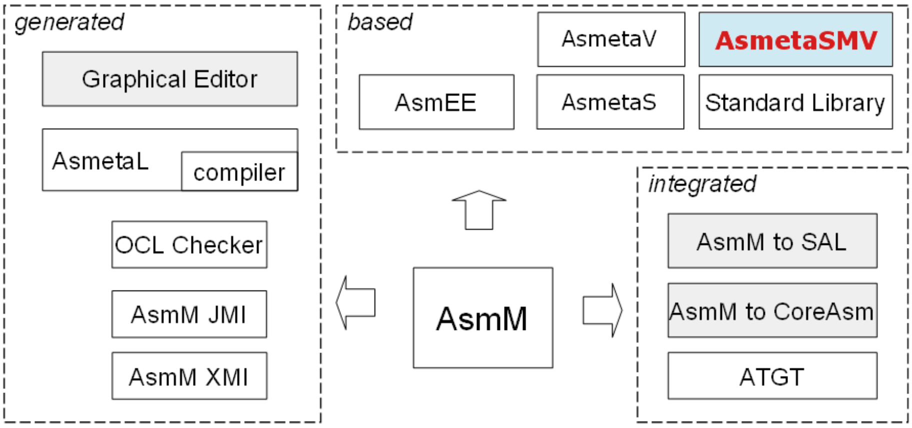


Figure 1.1: Toolset ASMETA

1.2 AsmetaSMV

AsmetaSMV is a based tool of the ASMETA toolset. Its aim is to enrich the ASMETA toolset with the capabilities of the model checker NuSMV; it translates a code written in AsmetaL into a NuSMV code.
The user can define the temporal properties he wants to check directly into the AsmetaL code; he could even don't know the NuSMV syntax, but just the AsmetaL one. The only thing a user must know to perform model checking over an AsmetaL code is, besides the AsmetaL language, the syntax of the temporal operators.
In the following sections we suppose that the reader knows the ASM theory [9], the AsmetaL language [10], the model checking theory [11] and the NuSMV language [12] (however in section 2 we give a brief introduction of NuSMV). The purpose of this text is to describe how to perform model checking over an AsmetaL code. We will describe which ASM elements are supported by the mapping and which are not; an ASM element is supported by the mapping if the tool is able to translate it into a NuSMV code. An ASM element, instead, could not be supported because of two reasons:

1. it's not possible to translate the element: the Integer domain, for example, cannot be mapped because NuSMV supports only finite types;

2. the mapping of the element would be too complicated: in future versions of the tool, many turbo rules could be mapped into NuSMV but, by now, are not supported by the tool.

Chapter 2

NuSMV

In this chapter we analyze the technique of *model checking* (section 2.1) and, in particular, of the model checker NuSMV (section 2.2). We describe only the concepts that are useful for the reading of the text; for a complete description of NuSMV you can see [12].

2.1 Model checking

Model checking is a formal verification technique; it permits to create abstract models of systems and verify that they satisfy properties defined in a temporal logic. In many situations the use of a model checker can be useful to the developers that, yet in the design phase, can discover possible errors of the model; in big projects, in fact, to discover a design error after the implementation phase can cause a loss of money and time.
A model checker works in three steps:

1. definition of a model $\mathcal{M}$ using the *Kripke structures* (section 2.1.1), a formalism similar to the finite state machines;

2. definition of a temporal formula ϕ, that describes a property that we want to verify (section 2.1.2);

3. the model checker verifies that $\mathcal{M} \vdash \phi$.

2.1.1 Kripke structure

A Kripke structure is define by the 4-uple

$$\mathcal{M} = (S, \Delta, S_0, L)$$

where:

- S is a finite set of states;

- Δ (or $\rightarrow$) is a transition total relation, that is
$$\forall s \in S \ \exists s' \in S \quad \text{such that} \quad s \rightarrow s'$$

- $S_0 \subseteq S$ is the set of initial states;

- $L : S \rightarrow 2^{AP}$ is a labelling function that links each state with a label; the label lists the atomic propositions that are true in that state. AP is a set of atomic propositions.

2.1.2 Computation Tree Logic (CTL)

Temporal logics are divided into:

- *Linear time logics* (LTL): they represent time as infinite sequences of instants; you can only declare properties that must be true over all sequences;

- *Branching Time Logics* (BTL): they represent time as a tree, where the root is the initial instant and its children the possible evolutions of the system; you can declare properties concerning all the paths or just some of them.

Temporal logics, moreover, can be classified in continuous time logics and discrete time logics.
In this text we will use *Computation Tree Logic* (CTL), a discrete time BTL. CTL permits to express logic formulas concerning paths, that is sequences of state transitions. Each CTL formula has a path quantifier that says if the formula must be true in all the paths (A, *along All paths*) or if must be true in at least one path (E, *Exists at least one path*). Moreover can be used the temporal operators:

- X p: the property p must be verified in the next state;

- F p: the property p must be verified in a future state;

- G p: the property p must be verified in all the states;

- p U q: the property p must be true until the q property becomes true.

It's $AP\{p, q, r, \ldots\}$ a set of atomic formulas; CTL formulas can be expressed in the following way:

$$\begin{aligned}\phi \ ::= & \top \mid \bot \mid p \in AP \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid AX\phi \mid EX\phi \mid A[\phi U \phi] \mid \\ & E[\phi U \phi] \mid AG\phi \mid EG\phi \mid AF\phi \mid EF\phi\end{aligned}$$

where $\top$, $\bot$, $\neg$, $\wedge$, $\vee$ and $\rightarrow$ are the connectives of propositional logic and AX, EX, AG, EG, AU, EU, AF and EF are temporal connectives.
The unary operators have the highest priority; then there are the binary operators $\vee$ and $\wedge$ and, at last, the binary operators $\rightarrow$, AU and EU.

2.2 NuSMV

NuSMV [12] is a symbolic model checker derived from CMU SMV [11]; it permits to verify properties written both in *Computation Tree Logic* (CTL) and in *Linear Temporal Logic* (LTL).
The internal representation of the model uses the *Binary Decision Diagrams* (BDDs), a particular type of graphs that permit to represent logic formulas in a compact and efficient way for the satisfiability analysis. A particular category of BDDs is used, the *Ordered Binary Decision Diagrams* (OBDDs), that permit to represent logic formulas in canonical form.
NuSMV is a transactional system in which the states are determined by the values of variables; transactions between the states are determined by the updates of the variables.
A NuSMV model is made of three principal sections:

- VAR: contains the declaration of variables;
- ASSIGN: contains the initialization (instruction *init*) and the update mechanism (instruction *next*) of the variables;
- SPEC: contains the CTL properties that must be verified by the model checker.

Code 2.1 is a small example of NuSMV model we will refer to in the following sections.

Code 2.1: NuSMV example

```
MODULE main
    VAR
        varBool: boolean;
        varNum: 1..5;
        varNumSet: {1, 3, 5};
        varEnum: {AA, BB, CC};
    ASSIGN
        init(varBool) := TRUE;
        init(varNum) := 1;
        init(varNumSet) := 1;
        next(varBool) := !varBool;
        next(varNum) := 2;
        next(varNumSet) :=
            case
                varNumSet = 5: 1;
                TRUE: varNumSet + 2;
            esac;
        next(varEnum) :=
            case
                varNumSet = 1: CC;
                varNumSet = 3: BB;
                varNumSet = 5: AA;
            esac;

SPEC    AG(varNumSet=1 <-> AX(varNumSet=3));
```

2.2.1 Variables

2.2.1.1 Variable type

Variables are declared in the VAR section with the specification of their types.
The type of a variable can be:

- *boolean*: it accepts as values the integers 0 and 1 or the equivalent literals FALSE and TRUE; variable *varBool* is a boolean variable;

- *Integer*; the variable can be defined:

  - over a values interval $a..b$ with $a < b$; such a variable is variable *varNum*;

  - over a set $\{a_i, \ldots, a_j\}$ of values not necessarily contiguous; a variable of such type is variable *varNumSet*;

- enumeration of symbolic constants; such a variable is variable *varEnum*.

2.2.1.2 Assign

The initialization and update instructions are executed in the ASSIGN section.

Initialization A variable *var* can be initialized to value v_0 with the instruction

$$\mathbf{init}(var) := v_0;$$

A variable can be either not initialized; an example of that is the variable *varEnum* of code 2.1. In such a situation, at the beginning NuSMV creates as many states as the number of values of the variable type; in each state the variable assumes a different value.

Update The value of a variable *var* in the next state is determined with the instruction

$$\mathbf{next}(var) := \ldots;$$

The next value can be determined in a straight way, as variables *varBool* and *varNum*, or in a conditional way through the **case** expression, as variables *varNumSet* and *varEnum*. In a **case** expression conditions are evaluated in sequence; the first true condition determines the resulting value; you can set, as last branch of the case instruction, the default condition *TRUE* that is selected if none of the previous conditions is safisfied. In code 2.1 the **next** instruction of *varNumSet* has the default value, the **next** instruction of *varEnum* doesn't. However we can notice that the conditions of the next value of variable *varEnum* are exhaustive and, so, there is no need of the default value. In code 2.2, instead, the conditions of the next value of *var* are not exhaustive.

Code 2.2: Not exhaustive conditions

```
MODULE main
    VAR
        var: 1..5;
    ASSIGN
        init(var) := 1;
        next(var) :=
            case
                var = 1: 3;
                var = 2: 5;
            esac;
```

The execution of code 2.2 signal, with an error message, the absence of exhaustiveness of the conditions:

```
[user@localhost code]$ NuSMV notExhaustive.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
```

```
file notExhaustive.smv: line 10: case conditions are not exhaustive


NuSMV terminated by a signal
```

2.2.2 Nondeterminism

In NuSMV it' possible to model non deterministic behaviours; they can be modeled in two ways:

- do not assign any value to a variable that, in such a way, can assume any value; this is the case of the missing initialization of variable *varEnum* in code 2.1;

- to assign to a variable a value randomly chosen from a set, through the expression $\{val_1, \ldots, val_n\}$; code 2.3 shows an example: in each transaction, the variable *var* can be 3 or 4.

Code 2.3: Nondeterminism

```
MODULE main
    VAR
        var: 1..5;

    ASSIGN
        init(var) := {3, 4};
        next(var) := {3, 4};
```

2.2.3 Invariant properties

It's possible to specify invariant conditions, that is properties that must be true in each state; the syntax of an invariant condition is

INVAR *boolExpr*;

where *boolExpr* is a boolean expression. In code 2.4 we can see how it's possible to reproduce the same semantic of code 2.3 through an invariant property.

Code 2.4: Invariant property

```
MODULE main
    VAR
        var: 1..5;
    INVAR (var=3 | var=4);
```

Even in this code, variable *var* can be only 3 or 4.

2.2.4 CTL properties

CTL properties are declared, through the keyword SPEC, in the following way

$$\textbf{SPEC}\ \ ctlForm;$$

where *ctlForm* is a CTL formula built in the way seen in section 2.1.2. In code 2.1 it's declared a property that verifies that, if in a state *varNumSet* is 1, in the next state is 3.

2.2.5 Example: Lift

In order to show an example of use of NuSMV, in this section we describe the NuSMV model for the lift problem; in the model we have declared some properties we want to verify.

Problem A lift connects four floors of a building, from the ground floor to the third floor. At each floor a button permits to request the lift. Inside the lift it's possible to select the destination floor. In the model, external callings executed from floor S_i and internal callings (executed in the lift) to stop at floor S_i are indiscernible: so we refer to a generic request from floor S_i. The lift must be used in a very tall building (hospitals, skyscrapers, etc.); the functioning cycle is the following:

- at the beginning the lift is at the ground floor and starts its travel towards the top;
- if during the ascent the lift receives a call from a floor S_u superior to the current position, when it reaches S_u stops (satisfies the request);
- when the lift arrives at the third floor, switches its direction and starts travelling towards the bottom;
- if during the descent the lift receives a call from a floor S_d inferior to the current position, when it reaches S_d stops (satisfies the request);
- when the lift arrives at the ground floor, switches its direction and starts travelling again towards the top.

We want that all the requests are satisfied and that there aren't situations in which the lift is in deadlock.
NuSMV code 2.5 contains the model of the problem.

Code 2.5: Lift

```
MODULE main
    VAR
        cabin: 0..3;
        dir: {up, down};
        request0: boolean;
        request1: boolean;
        request2: boolean;
        request3: boolean;
    ASSIGN
        init(cabin) := 0;
        init(dir) := up;
        init(request0) := FALSE;
        init(request1) := FALSE;
        init(request2) := FALSE;
        init(request3) := FALSE;
        next(cabin) :=
            case
                dir=up & cabin<3: cabin + 1; --ascent
                dir=down & cabin>0: cabin - 1; --descent
                TRUE: cabin;
            esac;
        next(dir) :=
            case
                dir=up & next(cabin)=3: down;
                dir=down & next(cabin)=0: up;
                TRUE: dir;
            esac;
        next(request0) :=
            case
                next(cabin)=0: FALSE; --the request is cleared
                request0: TRUE; --the request is kept
                TRUE: {FALSE, TRUE}; --can decide to make the request
            esac;
        next(request1) :=
            case
                next(cabin)=1: FALSE; --the request is cleared
                request1: TRUE; --the request is kept
                TRUE: {FALSE, TRUE}; --can decide to make the request
            esac;
        next(request2) :=
            case
                next(cabin)=2: FALSE; --the request is cleared
                request2: TRUE; --the request is kept
                TRUE: {FALSE, TRUE}; --can decide to make the request
            esac;
        next(request3) :=
            case
                next(cabin)=3: FALSE; --the request is cleared
                request3: TRUE; --the request is kept
                TRUE: {FALSE, TRUE}; --can decide to make the request
            esac;

--deadlock absence
SPEC    AG(EX(TRUE));

--safety properties
SPEC    AG!(dir=up & cabin=3)
SPEC    AG!(dir=down & cabin=0)

--liveness properties
```

```
SPEC    AG(request0 -> AF!request0);
SPEC    AG(request1 -> AF!request1);
SPEC    AG(request2 -> AF!request2);
SPEC    AG(request3 -> AF!request3);
```

Variable *cabin* records the current floor of the lift; variable *dir* records the direction of the lift; variables *request*0, *request*1, *request*2 and *request*3 are boolean variables that model the requests from the four floors. We hypothesize that each transaction corresponds to the passage from a floor to the next.

At the beginning the lift is at the ground floor with direction towards the top; there are no requests.

Let's see the update mechanism of variables (instructions *next*). Variable *cabin* is incremented when the lift is going towards the top, decremented when it's going towards the bottom. Variable *dir* is modified at the ground floor, where it becomes *up*, and at the third floor, where it becomes *down*. The four request variables are modified in the following way:

- if there is a request for a floor and the lift is arrived at that floor, the request is cleared;
- if there is a request for a floor and the lift is not yet arrived at that floor, the request is kept;
- if there is no request for a floor, nondeterministically a request could be executed for that floor.

Let's see some properties we want to verify.

First of all we want to verify that our model doesn't contains a *deadlock*, that is a state in which the system is blocked. The *deadlock absence* is verified through the property

```
AG(EX(TRUE))
```

that says that, for each state (AG), there is always at least a next state (EX).

Let's now declare some *safety* properties, that is properties that must always (or never) verified. The two properties

```
AG!(dir=up & cabin=3)
AG!(dir=down & cabin=0)
```

check that it's not possible (AG!) that the lift is going towards the top if it's at the third floor or that is going towards the bottom if it's at the ground floor. Finally we declare some *liveness* properties, that is properties that verify that some states are reached. The four properties

```
AG(request0 -> AF!request0)
AG(request1 -> AF!request1)
AG(request2 -> AF!request2)
AG(request3 -> AF!request3)
```

check that, if there is a request from a floor, the request sooner or later (AF) will be satisfied. The NuSMV code execution verifies the properties correctness:

```
[user@localhost code]$ NuSMV lift.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG (EX TRUE)  is true
-- specification AG !(dir = up & cabin = 3)  is true
-- specification AG !(dir = down & cabin = 0)  is true
-- specification AG (request0 -> AF !request0)  is true
-- specification AG (request1 -> AF !request1)  is true
-- specification AG (request2 -> AF !request2)  is true
-- specification AG (request3 -> AF !request3)  is true
```

Chapter 3

Supported ASM elements

3.1 Domains

An AsmetaL code that must be mapped into NuSMV can contain only domains that have a corresponding type in NuSMV.
The only supported domains are: Boolean, Enum domains, Concrete domains whose type domains are Integer or Natural.
Code 3.1 contains six functions of arity zero whose codomains are six different domains.

Code 3.1: Domains: AsmetaL model

```
asm domains
import ./StandardLibrary

signature:
    enum domain EnumDom = {EL_A | EL_B | EL_C}
    domain ConcrDomI1 subsetof Integer
    domain ConcrDomI2 subsetof Integer
    domain ConcrDomN1 subsetof Natural
    domain ConcrDomN2 subsetof Natural
    dynamic controlled fooB: Boolean
    dynamic controlled fooE: EnumDom
    dynamic controlled fooCI1: ConcrDomI1
    dynamic controlled fooCI2: ConcrDomI2
    dynamic controlled fooCN1: ConcrDomN1
    dynamic controlled fooCN2: ConcrDomN2

definitions:
    domain ConcrDomI1 = {1..5}
    domain ConcrDomI2 = {1, 3, 7}
    domain ConcrDomN1 = {2n..6n}
    domain ConcrDomN2 = {3n, 1n, 8n, 12n}
```

Code 3.2 shows the result of the mapping.

Code 3.2: Domains: NuSMV model

```
MODULE main
```

```
    VAR
        fooB: boolean;
        fooCI1: 1..5;
        fooCI2: {1, 3, 7};
        fooCN1: 2..6;
        fooCN2: {1, 12, 3, 8};
        fooE: {EL_A, EL_B, EL_C};
```

We can see that for each function a NuSMV variable has been created: in section 3.2 we'll describe exactly how it's executed the mapping of a function.
Now we are interested in the mapping of the domains. It's clear that the mapping of the Boolean domain and of the enum domain (*EnumDom*) is straightforward: in NuSMV there are both boolean and symbolic enum types. Concrete domain of Integer (*ConcrDomI1*, *ConcrDomI2*) and Natural (*ConcrDomN1*, *ConcrDomN2*), instead, become integer enums in NuSMV.

3.2 Functions

An ASM function, in order to be mapped into NuSMV, must be decomposed into its 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. The codomain of a function, instead, determines the type of the variable. Code 3.3 contains three functions of arity 1.

Code 3.3: Function of arity 1: AsmetaL model

```
asm arity1
import ./StandardLibrary

signature:
    domain SubDom subsetof Integer
    enum domain EnumDom = {AA | BB}
    dynamic controlled fooB: Boolean -> EnumDom
    dynamic controlled fooE: EnumDom -> SubDom
    dynamic controlled fooS: SubDom -> Boolean

definitions:
    domain SubDom = {1..2}

    main  rule r_Main =
        skip
```

Code 3.4 is the result of the translation.

Code 3.4: Function of arity 1: NuSMV model

```
MODULE main
    VAR
        fooB_FALSE: {AA, BB};
        fooB_TRUE: {AA, BB};
        fooE_AA: 1..2;
        fooE_BB: 1..2;
        fooS_1: boolean;
        fooS_2: boolean;
```

As we can see, for each AsmetaL function two NuSMV variables have been created; in fact, since all functions domains have two elements, each function has two locations.
Variables name is built in the following way:

$$idFunc\_elDom$$

where *idFunc* is the function name and *elDom* is an element of the function domain. In the following table we show the mapping of the three functions:

AsmetaL function	**AsmetaL locations**	**NuSMV variables**

fooB($b in Boolean)	fooB(false)
	fooB(true)
		fooB_FALSE
		fooB_TRUE

fooE($e in EnumDom)	fooE(AA)
	fooE(BB)
		fooE_AA
		fooE_BB

fooS($s in SubDom)	fooS(1)
	fooS(2)
		fooS_1
		fooS_2

Functions domains whose arity is greater than one must be Product domain, that is the cartesian product of a domains set.
The Product domain syntax is:

$$Prod(d_1,\ d_2, \ldots,\ d_n)$$

where $d_1, \ldots,\ d_n$ are the domains involved in the cartesian product.
Code 3.5 contains a function of arity two ($foo2$) and a function of arity three ($foo3$).

Code 3.5: Function of arity 2 and 3: AsmetaL model

```
asm arity2and3
import ./StandardLibrary

signature:
    domain SubDom subsetof Integer
    enum domain EnumDom = {AA | BB}
    dynamic controlled foo2: Prod(Boolean, EnumDom) -> SubDom
    dynamic controlled foo3: Prod(SubDom, EnumDom, SubDom) -> Boolean

definitions:
    domain SubDom = {1..2}

    main  rule r_Main =
        skip
```

Code 3.6 is the result of the translation.

Code 3.6: Function of arity 2 and 3: NuSMV model

```
MODULE main
    VAR
        foo2_FALSE_AA: 1..2;
        foo2_FALSE_BB: 1..2;
        foo2_TRUE_AA: 1..2;
        foo2_TRUE_BB: 1..2;
        foo3_1_AA_1: boolean;
        foo3_1_AA_2: boolean;
        foo3_1_BB_1: boolean;
        foo3_1_BB_2: boolean;
        foo3_2_AA_1: boolean;
        foo3_2_AA_2: boolean;
        foo3_2_BB_1: boolean;
        foo3_2_BB_2: boolean;
```

As we can see, the number of variables is equal to the product of the cardinality of the domains involved in the Product domain.
Given a function $func$ with domain $Prod(D_1, \ldots, D_n)$, the variables name is:

$$func\_elDom_1\_\ldots\_elDom_n$$

where $elDom_1 \in D_1, \ldots, elDom_n \in D_n$.
In the following table the mapping of the two functions is shown:

AsmetaL function	**AsmetaL locations**	**NuSMV variables**
foo2($b in Boolean, $e in EnumDom)	foo2(false, AA)	foo2_FALSE_AA
	foo2(false, BB)	foo2_FALSE_BB
	foo2(true, AA)	foo2_TRUE_AA
	foo2(true, BB)	foo2_TRUE_BB
foo3($i in SubDom, $e in EnumDom, $j in SubDom)	foo3(1, AA, 1)	foo3_1_AA_1
	foo3(1, AA, 2)	foo3_1_AA_2
	foo3(1, BB, 1)	foo3_1_BB_1
	foo3(1, BB, 2)	foo3_1_BB_2
	foo3(2, AA, 1)	foo3_2_AA_1
	foo3(2, AA, 2)	foo3_2_AA_2
	foo3(2, BB, 1)	foo3_2_BB_1
	foo3(2, BB, 2)	foo3_2_BB_2

3.2.1 Dynamic functions

AsmetaSMV supports only controlled and monitored dynamic functions. Before describing these functions, let's see a construction that is not supported by AsmetaSMV.
In AsmetaL it's possible that a location is determined using as an argument of the function another function; code 3.7 shows an example.

Code 3.7: Dynamic function as argument of another function

```
asm functionAsArg
import ./StandardLibrary

signature:
    enum domain EnumDom = {AA | BB | CC}
    dynamic monitored monArg: EnumDom
    dynamic controlled contrArg: EnumDom
    dynamic controlled foo: EnumDom -> Boolean
    dynamic controlled foo2: EnumDom -> Boolean

definitions:

    main  rule r_Main =
        par
            contrArg := AA

        //Not supported by AsmetaSMV
            foo(monArg) := true
        //Not supported by AsmetaSMV
            foo2(contrArg) := true
        endpar

default init s0:
    function contrArg = BB
```

Such constructions are not supported by AsmetaSMV, because the tool is not able to discover what NuSMV variable corresponds to the AsmetaL location.

3.2.1.1 Controlled functions

Controlled functions are the only functions whose value can be updated in a transaction rule. The initialization and the update of a dynamic location are mapped in the ASSIGN section through the *init* and *next* instructions. Code 3.8 contains the function *foo* whose locations are initialized and updated.

Code 3.8: Update: AsmetaL model

```
asm simpleUpdate
import ./StandardLibrary

signature:
    enum domain EnumDom = {AA | BB | CC}
    dynamic controlled foo: Boolean -> EnumDom
```

```
definitions:

    main  rule r_Main =
        par
            foo(true) := AA
            foo(false) := CC
        endpar

default init s0:
    function foo($b in Boolean) = BB
```

Code 3.9 is the result of the translation.

Code 3.9: Update: NuSMV model

```
MODULE main
    VAR
        foo_FALSE: {AA, BB, CC};
        foo_TRUE: {AA, BB, CC};
    ASSIGN
        init(foo_FALSE) := BB;
        init(foo_TRUE) := BB;
        next(foo_FALSE) := CC;
        next(foo_TRUE) := AA;
```

In an ASM model the update of a location could be guarded by a boolean condition; in NuSMV the next value of a variable can be guarded by the case expression. In code 3.10 the update of location *foo* to value *AA* is guarded by the condition *mon*.

Code 3.10: Guarded update: AsmetaL model

```
asm condUpdate
import ./StandardLibrary

signature:
    enum domain EnumDom = {AA | BB | CC}
    dynamic controlled foo: EnumDom
    dynamic monitored mon: Boolean

definitions:
    main  rule r_Main =
        if(mon) then
            foo := AA
        endif

default init s0:
    function foo = BB
```

Code 3.11 is the result of the translation.

Code 3.11: Guarded update: NuSMV model

```
MODULE main
    VAR
        foo: {AA, BB, CC};
        mon: boolean;
    ASSIGN
        init(foo) := BB;
        next(foo) :=
            case
                next(mon): AA;
                TRUE: foo;
            esac;
```

In NuSMV the variable *foo* is updated to value *AA* if the next value of *mon* is true[1].

The case expression contains also, at the end, a default condition (TRUE): if none of the previous conditions is satisfied, the value of the variable is set at the current value (the variable maintains its value).

Global update set During an ASM run, in each transition, the system builds the update set, that is the set of updates (couples location-value) that can be fired. AsmetaSMV, in order to translate the AsmetaL code into a NuSMV code, calculates all the update sets that can be generated during the run of an ASM. These update sets are merged into a *global update set*; the "global" update set lists, for each location, all the values to which the location can be updated. Each value is associated with the boolean condition that must be satisfied in order to execute the update. Let's see the update sets and the "global" update set of AsmetaL code 3.12.

Code 3.12: Example update: AsmetaL model

```
asm update
import ./StandardLibrary

signature:
    enum domain EnumDom = {AA | BB | CC}
    dynamic monitored mon: Boolean
    dynamic controlled foo: EnumDom
    dynamic controlled foo1: EnumDom

definitions:
    main  rule r_Main =
        if(mon) then
            par
                foo := AA
                foo1 := CC
            endpar
        else
            par
                foo := BB
                foo1 := AA
            endpar
        endif
```

There are two update sets that can be fired during the run of the ASM . If the value of monitored function *mon* is *true* the update set is:

[1] It's important to notice that we must check the next value of *mon*; this fact will be more clear in section 3.2.1.2

Location	Value
foo	AA
foo1	CC

If the value of *mon* is *false*, instead, the update set is:

Location	Value
foo	BB
foo1	AA

The "global" update set, that is the merge of the two update sets, is:

Location	Condition	Value
foo	mon	AA
	!mon	BB
foo1	mon	CC
	!mon	AA

Code 3.13 is the translation of AsmetaL code 3.12.

Code 3.13: Example update: NuSMV model

```
MODULE main
    VAR
        foo: {AA, BB, CC};
        foo1: {AA, BB, CC};
        mon: boolean;
    ASSIGN
        next(foo) :=
            case
                next(mon): AA;
                !(next(mon)): BB;
                TRUE: foo;
            esac;
        next(foo1) :=
            case
                next(mon): CC;
                !(next(mon)): AA;
                TRUE: foo1;
            esac;
```

As we have seen previously, the "global" update set is reported in the ASSIGN section; for each variable (location in AsmetaL), the next value is determined with the case expression where each value is associated with its condition.
It's important to underline that NuSMV doesn't resolve the problem of inconsistent updates; if the AsmetaL model contains an inconsistent update, also the NuSMV model will contain it. Code 3.14 contains an inconsistent update.

Code 3.14: Inconsistent update example: AsmetaL model

```
asm notConsistent
import ./StandardLibrary

signature:
    enum domain EnumDom = {AA | BB | CC}
    dynamic monitored mon: Boolean
    dynamic monitored mon2: Boolean
    dynamic controlled foo: EnumDom

definitions:

    main  rule r_Main =
        par
            if(mon != mon2) then
                foo := AA
            endif
            if(mon2 != mon) then
                foo := BB
            endif
        endpar
```

If, during a run of the model, the two monitored locations mon and $mon2$ have two different values, the simulator stops the execution and signals the inconsistent update. Code 3.15 is the result of the translation.

Code 3.15: Inconsistent update example: NuSMV model

```
MODULE main
    VAR
        foo: {AA, BB, CC};
        mon: boolean;
        mon2: boolean;
    ASSIGN
        next(foo) :=
            case
                (next(mon2) != next(mon)): BB;
                (next(mon) != next(mon2)): AA;
                TRUE: foo;
            esac;
```

NuSMV, instead, during the execution of the model doesn't signal any error; if monitored variables mon and $mon2$ are different in the next state, variable foo assumes the value BB, that is the value associated with the first satisfied condition. So, before using AsmetaSMV, we must be sure that the AsmetaL model doesn't contains any inconsistent updates.

3.2.1.2 Monitored functions

Monitored functions are functions whose value is set by the environment. In NuSMV, monitored variables are declared but they are neither initialized nor updated.

When NuSMV meets a monitored variable it creates a state for each value of the variable; code 3.16 contains the monitored variable *mon* that can assume four different values.

Code 3.16: Monitored variable in NuSMV

```
MODULE main
    VAR
        mon: 1..4;
```

Let's execute NuSMV with the option "-r" that prints the number of reachable states:

```
[user@localhost tosmv]$ NuSMV -r numStatiMon.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
system diameter: 1
reachable states: 4 (2^2) out of 4 (2^2)
```

We can see that there are four reachable states that correspond to the four values that the variable *mon* can assume.
It's important to describe how the monitored variables are used in a NuSMV model. When a monitored variable is used in the ASSIGN section (this means that, in AsmetaL, the corresponding monitored location it's used in a transition rule) its value is obtained through the next expression. Let's see an example (code 3.17).

Code 3.17: Monitored function - AsmetaL

```
asm mon
import ./StandardLibrary
import ./CTLLibrary

signature:
    dynamic monitored mon: Boolean
    dynamic controlled foo: Boolean

definitions:

    //axiom for simulation
    axiom over foo: foo = mon

    //property to translate into NuSMV
    //axiom over foo: ag(foo = mon)

    main  rule r_Main =
        foo := mon

default init s0:
    function foo = mon
```

Thanks to the axiom, during the simulation of the AsmetaL model, we can check that the controlled function *foo* is always (in each state) equal to the

monitored function *mon*. In fact, we must remember that the values of monitored locations are set at the beginning of the transaction, that is before the execution of the transition rules (in this case an update rule); this means that transition rules deal with the monitored locations values of the current state and not of the previous one.
A CTL property, equivalent to the axiom, has been written to check that NuSMV model keeps the same behaviour of the AsmetaL model.
We could think that the correct translation into NuSMV should be that shown in code 3.18.

Code 3.18: Monitored function - Wrong NuSMV code

```
MODULE main
    VAR
        foo: boolean;
        mon: boolean;
    ASSIGN
        init(foo) := mon;
        next(foo) := mon;
SPEC    AG(foo = mon);
```

But in code 3.18 the variable *foo* assumes, in the next state, the value of the variable *mon* in the current state: that is not the desired behaviour.
If we run NuSMV, in fact, we find a counterexample to the specification.

```
[user@localhost code]$ NuSMV monWrong.smv
*** This is NuSMV 2.4.1 (compiled on Sat Jun 13 10:57:42 UTC 2009)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

-- specification AG foo = mon  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  foo = 1
  mon = 1
-> Input: 1.2 <-
-> State: 1.2 <-
  mon = 0
```

NuSMV shows a state where *mon* is not equal to *foo*.
Code 3.19 shows the correct translation of AsmetaL code 3.17.

Code 3.19: Monitored function - Correct NuSMV code

```
MODULE main
    VAR
        foo: boolean;
        mon: boolean;
    ASSIGN
```

```
        init(foo) := mon;
        next(foo) := next(mon);
SPEC    AG(foo = mon);
```

Now, if we run NuSMV, we can see that the specification is satisfied.

```
[user@localhost code]$ NuSMV mon.smv
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification AG foo = mon  is true
```

3.2.1.3 Static and derived functions

Static and derived functions cannot be updated neither by an update rule nor by the environment; their value is set in the *definitions* section and never changes during the execution of the machine. Static functions do not depend on the state machine, derived functions, instead, do. AsmetaSMV doesn't distinguish between static and derived functions: their mapping is the same. In NuSMV static and derived functions are expressed through the DEFINE statement. Code 3.20 contains a static and a derived function.

Code 3.20: Static and derived functions: AsmetaL model

```
asm staticDerived
import ./StandardLibrary

signature:
    domain MyDomain subsetof Integer
    dynamic monitored mon1: Boolean
    dynamic monitored mon2: Boolean
    static stat: MyDomain
    derived der: Boolean

definitions:
    domain MyDomain = {1..4}

    function stat = 2
    function der = mon1 and mon2

    main rule r_Main =
        skip
```

Code 3.21 is the result of the translation.

Code 3.21: Static and derived functions: NuSMV model

```
MODULE main
```

```
    VAR
        mon1: boolean;
        mon2: boolean;
    DEFINE
        stat:= 2;
        der:= (mon1 & mon2);
```

Static function *stat* and derived function *der* have been mapped into two definitions in the NuSMV code.
To obtain a correct NuSMV code, the static and derived functions must be fully specified (i.e. specified in all the states of the machine). Let's see AsmetaL code 3.22.

Code 3.22: Not exhaustive derived function: AsmetaL model

```
asm derivedNotExhaustive
import ./StandardLibrary

signature:
    domain MyDomain subsetof Integer
    dynamic controlled foo : MyDomain
    dynamic monitored mon1: Boolean
    dynamic monitored mon2: Boolean
    derived der: Boolean

definitions:
    domain MyDomain = {1..4}

    function der =
        if(mon1) then
            if(mon2) then
                true
            else
                false
            endif
        endif

    main rule r_Main =
        if(der) then
            foo := 1
        endif
```

Derived function *der* is not defined when $mon1$ is *false*; in that case, during the valuation of *der*, simulator throws an exception and stop the simulation. Code 3.23 is the translation of code 3.22.

Code 3.23: Not exhaustive derived function: NuSMV model

```
MODULE main
    VAR
        foo: 1..4;
        mon1: boolean;
        mon2: boolean;
    DEFINE
        der:=
            case
                (mon1) & (!(mon2)): FALSE;
                (mon1) & (mon2): TRUE;
            esac;
    ASSIGN
        next(foo) :=
            case
                der & 1 in 1..4: 1;
                TRUE: foo;
            esac;
```

The execution of NuSMV code gives the following error:

```
[user@localhost tosmv]$ NuSMV derivedNotExhaustive.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

file derivedNotExhaustive.smv: line 17:
type error: value = FAILURE("case conditions are not exhaustive", line 11)
Expected a boolean expression


NuSMV terminated by a signal
```

NuSMV signals that the conditions of definition *der* are not exhaustive.

3.3 Rules

In this section we'll describe which rules are supported by AsmetaSMV.

3.3.1 Mapping process

Let's now briefly describe how it works the translation of the rules:

- the tool starts the translation in the main rule and continues executing a depth visit of the rules it encounters;
- the tool pushes the boolean conditions it encounters (e.g. if, switch, ...) on the global stack *Conds*; it removes the condition from stack *Conds* when it leaves the scope of the condition;
- when the tool encounters a location update, memorizes it in the "global" update set (section 3.2.1.1) with the right condition: the condition that must be satisfied, in order to perform the update, is the logical product of the conditions of stack *Conds*.

Let's see how it works the built of stack *Conds* and of the "global" update set over AsmetaL code 3.24.

Code 3.24: "Stack of conditions" example

```
asm stackConds
import ./StandardLibrary

signature:
    enum domain EnumDom = {AA | BB | CC}
    dynamic monitored mon: Boolean
    dynamic monitored mon2: Boolean
    dynamic controlled foo: EnumDom
    dynamic controlled foo1: EnumDom

definitions:
    main  rule r_Main =
        par
            foo1 := AA
            if(mon) then
                if(mon2) then
                    foo := BB
                else
                    foo := AA
                endif
            endif
        endpar
```

In the following table we show the contents of the stack and of the "global" update set during the visit of the AsmetaL model; the order of the rules in the table reflects the order of visit.

<table>
<tr><th>Rule</th><th>Stack</th><th>"Global" update set</th></tr>
<tr><td>foo1 := AA</td><td><table><tr><td></td></tr><tr><td></td></tr><tr><td>TRUE</td></tr></table></td><td><table><tr><th>Location</th><th>Condition</th><th>Value</th></tr><tr><td>foo1</td><td>TRUE</td><td>AA</td></tr></table></td></tr>
<tr><td>foo := BB</td><td><table><tr><td>mon2</td></tr><tr><td>mon</td></tr><tr><td>TRUE</td></tr></table></td><td><table><tr><th>Location</th><th>Condition</th><th>Value</th></tr><tr><td>foo</td><td>TRUE and mon and mon2</td><td>BB</td></tr><tr><td>foo1</td><td>TRUE</td><td>AA</td></tr></table></td></tr>
<tr><td>foo := AA</td><td><table><tr><td>!mon2</td></tr><tr><td>mon</td></tr><tr><td>TRUE</td></tr></table></td><td><table><tr><th>Location</th><th>Condition</th><th>Value</th></tr><tr><td>foo</td><td>TRUE and mon and !mon2</td><td>AA</td></tr><tr><td></td><td>TRUE and mon and mon2</td><td>BB</td></tr><tr><td>foo1</td><td>TRUE</td><td>AA</td></tr></table></td></tr>
</table>

When the tool encounters the update of location $foo1$, the stack contains only the default condition $TRUE$; the update is recorded in the update set

with the condition $TRUE$.
When it encounters the first update of location foo, the stack contains two more conditions, mon and $mon2$, belonging to the two nested **if** rules that precede the update. The update of foo to value BB is recorded with the condition $TRUE$ *and* mon *and* $mon2$.
When the tool encounters the second update of foo, the stack contains conditions $TRUE$, mon and $!mon2$ (because it's in the else branch of the second **if**); the update of foo to value AA is recorded with condition $TRUE$ *and* mon *and* $!mon2$.
We can notice that, when the tool visits the second update of foo, on the stack there is no more the condition mon: when the tool leaves the then branch, the condition has been removed from the stack. So a condition remains on the stack only during the visit of the rules that are in its visibility scope.
We have described in detail this little example to introduce the concepts that guide the mapping of all the rules.

Supported rules Supported rules are: update rule, macrocall rule, block rule, conditional rule, case rule, let rule, forall rule, choose rule. In the next section we describe the translation of all of them, except for the macrocall rule and the let rule: their mapping is trivial.

3.3.2 Update rule

The update rule syntax is:

$$l := t$$

where l is a location and t a term.
All the updates of an AsmetaL model are collected in the "global" update set that is reported in the ASSIGN section of the NuSMV model.

3.3.3 Block rule

The block rule syntax is:

$$\begin{array}{l}\textbf{par}\\ \qquad R_1\\ \qquad R_2\\ \qquad \dots\\ \qquad R_n\\ \textbf{endpar}\end{array}$$

where R_1, $R_2, \ldots,$ R_n are transition rules. In a block rule all the rules R_1, $R_2, \ldots,$ R_n are executed in parallel.
AsmetaSMV translates each rule individually. The contents of the stack *Conds*, at the beginning of each rule, is always the same.

3.3.4 Conditional rule

The conditional rule syntax is:

$$\begin{array}{l} \textbf{if}\ \mathit{cond}\ \textbf{then} \\ \qquad R_{then} \\ \textbf{else} \\ \qquad R_{else} \\ \textbf{endif} \end{array}$$

where *cond* is a boolean condition and R_{then} and R_{else} are transition rules. It's executed R_{then} if *cond* is true, R_{else} otherwise.
The translation process into NuSMV is:

- *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;
- *cond* is removed from stack *Conds*.
- If else branch is not null:
  - condition *notCond* (with $notCond := !cond$) 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;
  - *notCond* is removed from stack *Conds*.

AsmetaL code 3.25 contains an example of conditional rule.

Code 3.25: Conditional rule example: AsmetaL model

```
asm conditionalRule
import ./StandardLibrary

signature:
    enum domain EnumDom = {AA | BB | CC}
    dynamic controlled guard: EnumDom
    dynamic controlled foo: EnumDom

definitions:

    main rule r_Main =
        if(guard = CC) then
            foo := AA
        else
```

```
            foo := BB
        endif

default init s0:
    function guard = CC
```

Code 3.26 is the translation into NuSMV of AsmetaL code 3.25.

Code 3.26: Conditional rule example: NuSMV model

```
MODULE main
    VAR
        foo: {AA, BB, CC};
        guard: {AA, BB, CC};
    ASSIGN
        init(guard) := CC;
        next(foo) :=
            case
                !(guard = CC): BB;
                guard = CC: AA;
                TRUE: foo;
            esac;
        next(guard) := guard;
```

We can see that, correctly, variable foo is updated to value AA only if $guard = CC$ is true (then branch); otherwise the variable is updated to BB (else branch).

3.3.5 Case rule

The case rule syntax is:

```
switch $t$
       case $t_1$ : $R_1$
       $\ldots$
       case $t_n$ : $R_n$
       [otherwise $R_{other}$]
endswitch
```

where $t, t_1, \ldots, t_n$ are terms and $R_1, \ldots, R_n, R_{other}$ are transition rules. The case rule is equal to the switch statement of Java.
For each branch, the translation process into NuSMV is:

- condition $t = t_i$ is put on stack $Conds$;
- rule R_i is visited;
- condition $t = t_i$ is removed from stack $Conds$.

If default branch is not null:

- condition $t! = t_1 \;\&\; \dots \;\&\; t! = t_n$ is added to stack *Conds* and rule R_{other} is visited;
- the previous condition is removed from stack *Conds*.

AsmetaL code 3.27 contains an example of case rule.

Code 3.27: Case rule example: AsmetaL model

```
asm caseRule
import ./StandardLibrary

signature:
    enum domain EnumDom = {AA | BB | CC | DD}
    dynamic controlled sw: EnumDom
    dynamic controlled foo: EnumDom

definitions:

    main rule r_Main =
        switch(sw)
            case AA:
                foo := CC
            case BB:
                foo := BB
            otherwise
                foo := AA
        endswitch

default init s0:
    function sw = CC
```

Code 3.28 is the translation into NuSMV of AsmetaL code 3.27.

Code 3.28: Case rule example: NuSMV model

```
MODULE main
    VAR
        foo: {AA, BB, CC, DD};
        sw: {AA, BB, CC, DD};
    ASSIGN
        init(sw) := CC;
        next(foo) :=
            case
                (sw != AA) & (sw != BB): AA;
                sw = BB: BB;
                sw = AA: CC;
                TRUE: foo;
            esac;
        next(sw) := sw;
```

The two case branches have been transformed into two equalities ($sw = AA$ and $sw = BB$). Otherwise branch has been transformed into the and of two disequalities ($sw! = AA \,\&\, sw! = BB$).

3.3.6 Forall rule

The forall rule syntax is:

$$\textbf{forall } v_1 \textbf{ in } D_1, \dots, v_n \textbf{ in } D_n \textbf{ with } G_{v_1,\dots,v_n} \textbf{ do}$$
$$R_{v_1,\dots,v_n}$$

where $v_1, \dots, v_n$ are variables and $D_1, \dots, D_n$ their domains. $G_{v_1,\dots,v_n}$ is a boolean condition over $v_1, \dots, v_n$. $R_{v_1,\dots,v_n}$ is a rule that contains occurrences of $v_1, \dots, v_n$.
The purpose of the forall rule is to execute the rule $R_{v_1,\dots,v_n}$ with all the values of variables $v_1, \dots, v_n$ that satisfy the condition $G_{v_1,\dots,v_n}$. The number nR of branches to evaluate is equal to the product of the cardinalities of domains $D_1, \dots, D_n$:

$$nR = \prod_{i=1}^{n} |D_i|$$

The translation process into NuSMV, for each values tuple $d_1^{j_1}$, ..., $d_n^{j_n}$ with $d_1^{j_1} \in D_1, \dots, d_n^{j_n} \in D_n$, executes the following operations:

- variables $v_1, \dots, v_n$ assume values $d_1^{j_1}$, ..., $d_n^{j_n}$;
- $G_{d_1^{j_1},\, \dots,\, d_n^{j_n}}$ and $R_{d_1^{j_1},\, \dots,\, d_n^{j_n}}$ are the condition and the rule where the variables have been replaced with the current values $d_1^{j_1}$, ..., $d_n^{j_n}$;
- condition $G_{d_1^{j_1},\, \dots,\, d_n^{j_n}}$ is put on stack *Conds*;
- rule $R_{d_1^{j_1},\, \dots,\, d_n^{j_n}}$ is visited;
- condition $G_{d_1^{j_1},\, \dots,\, d_n^{j_n}}$ is removed from stack *Conds*.

AsmetaL code 3.29 contains an example of forall rule.

Code 3.29: Forall rule example: AsmetaL model

```
asm forallRule
import ./StandardLibrary

signature:
    domain ConcrDom subsetof Integer
    dynamic controlled foo: ConcrDom -> ConcrDom

definitions:
    domain ConcrDom = {1..4}

    main rule r_Main =
        forall $x in ConcrDom with $x < 3 do
            foo($x) := 1

default init s0:
    function foo($x in ConcrDom) = $x
```

Code 3.30 is the translation into NuSMV of AsmetaL code 3.29.

Code 3.30: Forall rule example: NuSMV model

```
MODULE main
    VAR
        foo_1: 1..4;
        foo_2: 1..4;
        foo_3: 1..4;
        foo_4: 1..4;
    ASSIGN
        init(foo_1) := 1;
        init(foo_2) := 2;
        init(foo_3) := 3;
        init(foo_4) := 4;
        next(foo_1) :=
            case
                (1 < 3) & 1 in 1..4: 1;
                TRUE: foo_1;
            esac;
        next(foo_2) :=
            case
                (2 < 3) & 1 in 1..4: 1;
                TRUE: foo_2;
            esac;
        next(foo_3) :=
            case
                (3 < 3) & 1 in 1..4: 1;
                TRUE: foo_3;
            esac;
        next(foo_4) :=
            case
                (4 < 3) & 1 in 1..4: 1;
                TRUE: foo_4;
            esac;
```

We can see that the forall rule has been decomposed into four instructions that corresponds to the four values of variable $$x$.

3.3.7 Choose rule

The choose rule syntax is:

$$\begin{array}{l} \textbf{choose } v_1 \textbf{ in } D_1, \ldots, v_n \textbf{ in } D_n \textbf{ with } G_{v_1,\ldots,v_n} \textbf{ do} \\ \qquad R_{v_1,\ldots,v_n} \\ \qquad [\textbf{ifnone } R_{ifnone}] \end{array}$$

where $v_1, \ldots, v_n$ are variables and $D_1, \ldots, D_n$ their domains. $G_{v_1,\ldots,v_n}$ is a boolean condition over $v_1, \ldots, v_n$. $R_{v_1,\ldots,v_n}$ is a rule that contains occurrences of $v_1, \ldots, v_n$.
The purpose of the choose rule is to execute one time the rule $R_{v_1,\ldots,v_n}$ with some variables $v_1, \ldots, v_n$ that satisfy $G_{v_1,\ldots,v_n}$. The number nR of branches to evaluate is equal to the product of the cardinalities of domains $D_1, \ldots, D_n$:

$$nR = \prod_{i=1}^{n} |D_i|$$

Optional branch **ifnone** contains the rule R_{ifnone} that must be executed if there aren't 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 the identifier *chId*.
In NuSMV, for each variable v_i, a variable $var\_v_i\_chId$ is created; the type of such variable is obtained with the mapping of domain D_i.
The translation process into NuSMV, for each values tuple $d_1^{j_1}, \dots, d_n^{j_n}$ with $d_1^{j_1} \in D_1, \dots, d_n^{j_n} \in D_n$, executes the following operations:

- variables $v_1, \dots, v_n$ assume values $d_1^{j_1}, \dots, d_n^{j_n}$;
- $G_{d_1^{j_1}, \dots, d_n^{j_n}}$ and $R_{d_1^{j_1}, \dots, d_n^{j_n}}$ are the condition and the rule where the variables have been replaced with the current values $d_1^{j_1}, \dots, d_n^{j_n}$;
- for each variable v_i, it's put on stack *Conds* the condition

$$var\_v_i\_chId = d_i^{j_i}$$

- condition $G_{d_1^{j_1}, \dots, d_n^{j_n}}$ is put on stack *Conds*;
- rule $R_{d_1^{j_1}, \dots, d_n^{j_n}}$ is visited;
- conditions on variables and $G_{d_1^{j_1}, \dots, d_n^{j_n}}$ are removed from stack *Conds*.
- If **ifnone** branch is not null:
  - it's put on stack *Conds* the condition

$$ifNoneCond = \bigwedge_{\substack{d_1^{j_1} \in D_1 \\ \dots \\ d_n^{j_n} \in D_n}} !G_{d_1^{j_1}, \dots, d_n^{j_n}}$$

    where the number of terms of the logical product is nR.
  - rule R_{ifnone} is visited;
  - the previous condition is removed from stack *Conds*.

To be sure that, in each state, variables $var\_v_i\_chId$ $(i = 1, \dots, n)$ assume a condition that satisfy $G_{v_1,\dots,v_n}$, we define the following invariant in the INVAR section:

$$\bigvee_{\substack{d_1^{j_1} \in D_1 \\ \dots \\ d_n^{j_n} \in D_n}} \Big(\big(var\_v_1\_idCh = d_1^{j_1} \;\&\; \dots \;\&\; var\_v_n\_idCh = d_n^{j_n}\big) \;\&\; G_{d_1^{j_1}, \dots, d_n^{j_n}} \Big) \;|$$
$$ifNoneCond$$

AsmetaL code 3.31 contains an example of choose rule.

Code 3.31: Choose rule example: AsmetaL model

```
asm chooseRule
import ./StandardLibrary

signature:
    domain MyDomain subsetof Integer
    dynamic controlled foo: MyDomain

definitions:
    domain MyDomain = {1..4}

    main rule r_Main =
        choose $x in MyDomain with $x < 2 do
                foo := $x + 2
            ifnone
                foo := 4
```

Code 3.32 is the translation into NuSMV of AsmetaL code 3.31.

Code 3.32: Choose rule example: NuSMV model

```
MODULE main
    VAR
        foo: 1..4;
        var_$x_0: 1..4;
    ASSIGN
        next(foo) :=
            case
                (var_$x_0 = 1) & (1 < 2) & (1 + 2) in 1..4: (1 + 2);
                (var_$x_0 = 2) & (2 < 2) & (2 + 2) in 1..4: (2 + 2);
                (var_$x_0 = 3) & (3 < 2) & (3 + 2) in 1..4: (3 + 2);
                (var_$x_0 = 4) & (4 < 2) & (4 + 2) in 1..4: (4 + 2);
                !(1 < 2) & !(2 < 2) & !(3 < 2) &
                !(4 < 2) & 4 in 1..4: 4;
                TRUE: foo;
            esac;
    INVAR ((var_$x_0 = 1) & (1 < 2)) | ((var_$x_0 = 2) & (2 < 2)) |
        ((var_$x_0 = 3) & (3 < 2)) | ((var_$x_0 = 4) & (4 < 2)) |
        (!(1 < 2) & !(2 < 2) & !(3 < 2) & !(4 < 2));
```

We can see that the choose rule has been decomposed into:

- four update instructions that correspond to the four values of variable $\$x$; each instruction is guarded by the condition of the choose rule where the variable $\$x$ has been replaced with its values;
- an update instruction that corresponds to the **ifnone** branch;
- an INVAR declaration.

Chapter 4

CTL properties

In this chapter we describe how to declare CTL properties in an AsmetaL model. In section 4.1 we see how to rewrite AsmetaL axiom so that they can be evaluated in NuSMV. In section 4.2, instead, we see how to declare a CTL property of whatever type.

4.1 Mapping of AsmetaL axioms

In AsmetaL, an axiom is a property that must be verified in each state of the machine; to be sure that the axiom is always true we should simulate the model as many times as the number of states.
An AsmetaL axiom, in order to be translated into NuSMV, must be written in a slightly different way; the axiom

$$\textbf{axiom over } id_1, \ldots, id_n : \quad ax_{id_1,\ldots,id_n} \tag{4.1}$$

must be rewritten in the following way

$$\textbf{axiom over } id_1, \ldots, id_n : \quad ag(ax_{id_1,\ldots,id_n}) \tag{4.2}$$

where ag is an AsmetaL function, equivalent to the temporal operator AG of NuSMV. ag function means that the property $ax_{id_1,\ldots,id_n}$ must be verified in all the states: that is just the purpose of an AsmetaL axiom.
An AsmetaL model containing properties like 4.2 cannot be simulated with AsmetaS: in fact the simulator cannot understand the meaning of ag function. If you want to simulate the model, before translating it in NuSMV, you should comment property 4.2 and reintroduce axiom 4.1.
In the AsmetaL code 4.1 an axiom verifies that location $fooA$ is different from $fooB$ in each state; in order to map the axiom in NuSMV we have declared

a CTL property in which the axiom is the argument of the *ag* function (the axiom has been commented).

Code 4.1: Axioms mapping: AsmetaL model

```
asm ag
import ./StandardLibrary
import ./CTLlibrary

signature:
    dynamic controlled fooA: Boolean
    dynamic controlled fooB: Boolean

definitions:

    //axiom for simulation with AsmetaS
    //axiom over fooA, fooB: fooA != fooB

    //property for NuSMV
    axiom over fooA, fooB: ag(fooA != fooB)

    main rule r_Main =
        par
            fooA := not(fooA)
            fooB := not(fooB)
        endpar

default init s0:
    function fooA = true
    function fooB = false
```

Code 4.2 is the NuSMV code obtained from the mapping of AsmetaL code 4.1.

Code 4.2: Axioms mapping: NuSMV model

```
MODULE main
    VAR
        fooA: boolean;
        fooB: boolean;
    ASSIGN
        init(fooA) := TRUE;
        init(fooB) := FALSE;
        next(fooA) := !(fooA);
        next(fooB) := !(fooB);
SPEC    AG(fooA != fooB);
```

Let's check the correctness of the property through the execution of the NuSMV code.

```
[user@localhost code]$ NuSMV ag.smv
*** This is NuSMV 2.4.1 (compiled on Sat Jun 13 10:57:42 UTC 2009)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

-- specification AG fooA != fooB  is true
```

4.2 Declaration of CTL properties

Let's now describe in details how to declare CTL properties in an AsmetaL model.
In AsmetaL, the syntax of an axiom is:

$$\textbf{axiom over } id_1, \ldots, id_n : \quad ax_{id_1,\ldots,id_n}$$

where $id_1, \ldots, id_n$ are names of domains, functions or rules; $ax_{id_1,\ldots,id_n}$ is a boolean expression containing occurrences of $id_1, \ldots, id_n$.
In NuSMV, CTL properties are declared through the keyword SPEC:

$$\textbf{SPEC} \quad p_{CTL}$$

where p_{CTL} is a CTL formula.
We have decided to let the user declare CTL properties in the axiom section of the AsmetaL model.
The syntax of a CTL property in AsmetaL is:

$$\textbf{axiom over } id_1, \ldots, id_n : \quad p_{CTL}^{id_1,\ldots,id_n}$$

where $id_1, \ldots, id_n$, as in a normal axiom, are names of domains, functions or rules; $p_{CTL}^{id_1,\ldots,id_n}$, instead, is a CTL formula containing occurrences of $id_1, \ldots, id_n$.
In order to write CTL formulas in AsmetaL, we have created the library *CTL-library.asm* where, for each CTL operator, an equivalent function is declared.
The following table shows all the CTL functions.

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

In order to use CTL functions in our AsmetaL model, the CTL library *CTL-library.asm* must be imported.
AsmetaL code 4.3 contains three CTL properties; the first two properties are true, the third, instead, is false.

Code 4.3: CTL properties: AsmetaL model

```
asm ctlExample
import ./StandardLibrary
import ./CTLlibrary

signature:
    dynamic controlled fooA: Boolean
    dynamic controlled fooB: Boolean
    dynamic monitored mon: Boolean

definitions:

    axiom over fooA: ag(fooA iff ax(not(fooA))) //true
    axiom over fooA: ag(not(fooA) iff ax(fooA)) //true
    //false. Gives counterexample.
    axiom over fooA, fooB: not(ef(fooA != fooB))

    main rule r_Main =
        par
            fooA := not(fooA)
            if(mon) then
                fooB := not(fooB)
            endif
        endpar

default init s0:
    function fooA = true
    function fooB = true
```

Code 4.4 is the translation into NuSMV of code 4.3.

Code 4.4: CTL properties: NuSMV model

```
MODULE main
    VAR
        fooA: boolean;
        fooB: boolean;
        mon: boolean;
    ASSIGN
        init(fooA) := TRUE;
        init(fooB) := TRUE;
        next(fooA) := !(fooA);
        next(fooB) :=
            case
                next(mon): !(fooB);
                TRUE: fooB;
            esac;
SPEC    AG(fooA <-> AX(!(fooA)));
SPEC    AG(!(fooA) <-> AX(fooA));
SPEC    !(EF(fooA != fooB));
```

Let's execute NuSMV code to verify the properties.

```
[user@localhost tosmv]$ NuSMV ctlExample.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
```

```
-- specification AG (fooA <-> AX !fooA)  is true
-- specification AG (!fooA <-> AX fooA)  is true
-- specification !(EF fooA != fooB)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  fooA = 1
  fooB = 1
  mon = 0
-> Input: 1.2 <-
-> State: 1.2 <-
  fooA = 0
```

We can see that the first two properties are true: variable *fooA*, in fact, changes its value at each step. The last property, instead, is false and NuSMV shows a counterexample: it exists a state where variables *fooA* and *fooB* are different (State: 1.2).

4.2.1 Monitored locations in CTL properties

When we write a CTL property concerning a monitored location, we should remember that the values of monitored locations are set at the beginning of the transaction, that is before the execution of the transition rules. This means that transition rules deal with the monitored locations values of the current state and not of the previous one. When a transition rule deals with a controlled location, instead, it reads the location value of the previous state.
AsmetaL code 4.5 shows an example that, thanks to three CTL properties, can explain more clearly the concepts previously described.

Code 4.5: Monitored locations in CTL properties: AsmetaL model

```
asm monitoredExample
import ./StandardLibrary
import ./CTLlibrary

signature:
    dynamic controlled foo: Boolean
    dynamic controlled fooA: Boolean
    dynamic controlled fooB: Boolean
    dynamic monitored mon: Boolean

definitions:
    axiom over foo, mon: ag(foo = mon) //true
    axiom over fooA, fooB: ag(fooA = fooB) //false
    axiom over fooA, fooB: (fooA = fooB) iff ax(ag(fooA != fooB)) //true

    main rule r_Main =
        par
```

```
            fooB := not(fooB)
            foo := mon
            fooA := fooB
        endpar

default init s0:
    function foo = mon
    function fooA = true
    function fooB = true
```

Code 4.6 contains the NuSMV translation of AsmetaL code 4.5.

Code 4.6: Monitored locations in CTL properties: NuSMV model

```
MODULE main
    VAR
        foo: boolean;
        fooA: boolean;
        fooB: boolean;
        mon: boolean;
    ASSIGN
        init(foo) := mon;
        init(fooA) := TRUE;
        init(fooB) := TRUE;
        next(foo) := next(mon);
        next(fooA) := fooB;
        next(fooB) := !(fooB);
SPEC      AG(foo = mon);
SPEC      AG(fooA = fooB);
SPEC      fooA = fooB <-> AX(AG(fooA != fooB));
```

The property
axiom over foo, mon: ag(foo = mon)
is verified. In fact the controlled location *foo*, in each state, is updated to the value of monitored location *mon*. The value of monitored location *mon* is set before the execution of the update.
The property
axiom over fooA, fooB: ag(fooA = fooB)
is not verified. The controlled location *fooA*, in each state, is updated to the value of controlled location *fooB*: it's important to notice that it's used the value of the previous state (and not of the current state like with the monitored location).
The property
axiom over fooA, fooB: (fooA = fooB) iff ax(ag(fooA != fooB))
is verified. In fact, locations *fooA* and *fooB* are equal only in the initial state and then are always different.

4.2.2 Choose rule behaviour

Now we want to warn the user about the behaviour of the choose rule in the NuSMV model[1]. In fact, the translation of a choose rule could result obscure to an occasional user.
The problem is the following:

- the choose rule variables become variables in NuSMV;
- the values of these variables are bounded by boolean conditions; these conditions could contains some other variables (locations in AsmetaL); so the choose variables values could depend on the values of some other variables;
- since the choose variables values are set with the INVAR instruction (invariant), they are updated at the end of the previous state and not at the beginning of the current one.

All these things create a strange (but correct) behaviour of the choose rules in NuSMV.
Let's see an example to better explain the problem.
AsmetaL code 4.7 contains a choose rule and a property that is certainly false.

Code 4.7: Choose behaviour: AsmetaL model

```
asm choose
import ./StandardLibrary
import ./CTLlibrary

signature:
    domain MyDomain subsetof Integer
    dynamic controlled foo: MyDomain

definitions:
    domain MyDomain = {1..4}

    //false. We want the counterexample.
    axiom over foo: ag(foo!=2)

    main rule r_Main =
        choose $x in MyDomain with $x > foo do
                foo := $x

default init s0:
    function foo = 1
```

Code 4.8 is the result of the translation of AsmetaL code 4.7.

[1] We describe it in this section because now we can use the NuSMV execution as a support of our explanation.

Code 4.8: Choose behaviour: NuSMV model

```
MODULE main
    VAR
        foo: 1..4;
        var_$x_0: 1..4;
    ASSIGN
        init(foo) := 1;
        next(foo) :=
            case
                ((var_$x_0 = 3) & (3 > foo)) & 3 in 1..4: 3;
                ((var_$x_0 = 1) & (1 > foo)) & 1 in 1..4: 1;
                ((var_$x_0 = 4) & (4 > foo)) & 4 in 1..4: 4;
                ((var_$x_0 = 2) & (2 > foo)) & 2 in 1..4: 2;
                TRUE: foo;
            esac;
    INVAR   ((var_$x_0 = 1) & (1 > foo)) | ((var_$x_0 = 2) & (2 > foo)) | ((
    var_$x_0 = 3) & (3 > foo)) |
        ((var_$x_0 = 4) & (4 > foo)) | ((!(1 > foo)) & (!(2 > foo)) & (!(3 >
     foo)) & (!(4 > foo)));
SPEC    AG(foo != 2);
```

Since the property is false, NuSMV gives us a counterexample. The counterexample shows the print of the path that takes to a state in which the property is false. In this way we can see how changes the value of variable *var_ $x_ 0* in the NuSMV model. Variable *var_ $x_ 0* is the NuSMV variable that models the AsmetaL variable $\$x$ of the choose rule.

```
[user@localhost tosmv]$ NuSMV choose.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG foo != 2  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  foo = 1
  var_$x_0 = 2
-> Input: 1.2 <-
-> State: 1.2 <-
  foo = 2
  var_$x_0 = 3
```

The property is false because it exists a state (State: 1.2) in which the variable *foo* is equal to 2. We could think that, in this state, the variable *var_ $x_ 0* should be equal to 2. The counterexample trace, instead, shows us that the variable *var_ $x_ 0* is equal to 2 in the previous state (State: 1.1). In fact, the update of variable *foo* to 2 is guarded by the value of the choose variable in the previous state.

Chapter 5

User guide

AsmetaSMV tool, as all ASMETA tools, can be downloaded by project site [1] through a SVN client. Once downloaded, the project *tosmv* must be imported in Eclipse [13].
By the project site, it can also be downloaded the executable jar archive *AsmetaSMV.jar* that permits to use the tool through the command line, without Eclipse.

5.1 Eclipse project

In this section we describe how to use the tool in Eclipse.
First of all we must import the project *tosmv* and the projects it needs, *parser*, *interpreter* and *libs* (figure 5.1).

Once imported, the projects must be compiled through the command *Project/Build All*[1]. Now the tool can be executed in the following way:

- visualize the contextual menu over *AsmetaSMV.java* file of *tosmv* project and select *Run As/Run configurations...* (figure 5.2(a));
- in the new window select *Java Application*;
- in section *Arguments* insert the file name and, eventually, the options described in section 5.1.2 (figura 5.2(b));
- select command *Run*.

[1] In Eclipse there is also the option *Build Automatically* that compiles automatically all the projects of the workspace.

If the mapping process ends correctly, in the directory that contains the input AsmetaL file (*example.asm*) there is the output NuSMV file with the same name of the AsmetaL file and extension ".smv" (*example.smv*). If you want, the tool can execute the NuSMV file and show the output on the standard output (section 5.1.2).

5.1.1 Executable jar archive

To use the tool through the command line, we must open a terminal and move to the directory of the project; then we must execute the tool in the following way:

```
java -jar AsmetaSMV.jar [-en] [-ns] [-nc] file.asm
```

where "-en" "-ns" and "-nc" are the execution options, described in section 5.1.2.

5.1.2 Execution options

The tool has three execution options:

- -en: the tool, after the translation, executes the NuSMV file and shows the result on the standard output (section 5.1.2.1);
- -ns: the tool doesn't simplify the boolean conditions (section 5.1.2.2);
- -nc: the tool doesn't insert the check on the domain of integer variables (section 5.1.2.3).

5.1.2.1 Embedded execution of NuSMV file

The execution option "-en" permits to run the NuSMV file just after the translation process. In this way, NuSMV execution is embedded in the execution of a Java program and so, in general, it will be slower than the execution of the same model in the standard way (execution of NuSMV in a terminal).
So, we suggest to use "-en" option when NuSMV model is not too big[2].
The embedded execution of NuSMV has an advantage. Let's see the execution of a NuSMV model (we don't care what model) in the standard way:

[2] We remember that the complexity of a NuSMV model depends by the number of variables; so, in our scenario, it depends by the number of locations of the AsmetaL model.

```
[user@localhost tosmv]$ NuSMV example.smv
** This is NuSMV 2.4.0 (compiled on Sat Oct 4 10:17:49 UTC 2008)
** For more information on NuSMV see <http://nusmv.irst.itc.it>
** or email to <nusmv-users@irst.itc.it>.
** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG foo_AA  is true
-- specification AG foo2_TRUE_BB  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  foo2_TRUE_BB = 0
  foo_AA = 1
  foo_BB = 1
```

Let's see now the execution of the same NuSMV model, but in the embedded way:

```
Execution of NuSMV code: ..
-------------------------------------
> NuSMV -dynamic -coi examples\example.smv
-------------------------------------
** This is NuSMV 2.4.0 (compiled on Sat Oct 4 10:17:49 UTC 2008)
** For more information on NuSMV see <http://nusmv.irst.itc.it>
** or email to <nusmv-users@irst.itc.it>.
** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG foo(AA)  is true
-- specification AG foo2(true,BB)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  foo2(true,BB) = 0
  foo(AA) = 1
  foo(BB) = 1
```

We see that, in the normal way, the variables names have the format previously described. In the embedded way, instead, the variables names are replaced by the corresponding AsmetaL locations names: for the user should be easier to read the result of the execution.

5.1.2.2 Simplification of boolean conditions

In order to reduce the complexity of NuSMV code, AsmetaSMV simplify, where possible, the boolean conditions memorized in the *update set*. Condition $1 < 3$, for example, is recognized as always true and so it's replaced with the literal $TRUE$. A literal $TRUE$ belonging to a logical product, instead, is removed because it's not determinant in the evaluation.

Asmetal code 5.1 contains some conditions that take part in the built of the *updateSet*.

Code 5.1: Boolean conditions: AsmetaL model

```
asm esSempl
import ./StandardLibrary

signature:
    domain MyDomain subsetof Integer
    dynamic controlled foo: MyDomain
    dynamic monitored mon: Boolean

definitions:
    domain MyDomain = {1..4}

    main  rule r_Main =
        if(1 < 2) then
            if(mon) then
                foo := 2
            else
                foo := 3
            endif
        else
            foo := 1
        endif
```

Code 5.2 is the translation in NuSMV of Asmetal code 5.1: no simplification has been made.

Code 5.2: Boolean conditions: NuSMV model without simplification

```
MODULE main
    VAR
        foo: 1..4;
        mon: boolean;
    ASSIGN
        next(foo) :=
            case
                (1 < 2) & next(mon) & 2 in 1..4: 2;
                !(1 < 2) & 1 in 1..4: 1;
                (1 < 2) & !next(mon) & 3 in 1..4: 3;
                TRUE: foo;
            esac;
```

We can see that conditions $(1 < 2)$ and $!(1 < 2)$ haven't been simplified. In code 5.3, instead, the boolean conditions have been simplified.

Code 5.3: Boolean conditions: NuSMV model with simplification

```
MODULE main
    VAR
        foo: 1..4;
        mon: boolean;
    ASSIGN
        next(foo) :=
            case
                next(mon) & 2 in 1..4: 2;
                !next(mon) & 3 in 1..4: 3;
                TRUE: foo;
```

```
             esac;
```

In the following table, we show the conditions associated to the *foo* update in code 5.2 and in code 5.3.

Code 5.2	**Code 5.3**	**Description**
$(1 < 2)$ & next(mon)	next(mon)	Condition $(1 < 2)$ has been removed because it's not determinant in the evaluation of the logical product.
$(1 < 2)$ & !next(mon)	!next(mon)	Condition $(1 < 2)$ has been removed because it's not determinant in the evaluation of the logical product.
!$(1 < 2)$	-	Condition !$(1 < 2)$ is always false; so, the update of *foo* to value 1 has been removed.

The tool, by default, executes the simplification; the execution option "-ns" permits not to execute the simplifications. Sometimes, in fact, it can be necessary to read the obtained NuSMV code. By experience, we can say that it's easier to read a NuSMV code without simplifications rather than one with simplifications; the first one, in fact, reflects better the structure of the original AsmetaL model; the latter one, instead, thanks to the simplifications could be much more different.

5.1.2.3 Check on integer domains

The syntax on an update rule is:

$$l := t$$

where l is a location and t a term.
The mapping of updates of integer locations introduces a problem. Let's see the AsmetaL code 5.4.

Code 5.4: Update rule 1: AsmetaL model

```
asm updateRule
import ./StandardLibrary

signature:
    domain MyDomain subsetof Integer
    dynamic controlled foo: MyDomain

definitions:
    domain MyDomain = {1..4}

    main  rule r_Main =
        foo := foo + 1

default init s0:
    function foo = 1
```

We could think that the correct translation in NuSMV should be that shown in code 5.5.

Code 5.5: Update rule 1: wrong NuSMV model

```
MODULE main
    VAR
        foo: 1..4;
    ASSIGN
        init(foo) := 1;
        next(foo) := foo + 1;
```

The execution of code 5.5 gives the following error:

```
[user@localhost tosmv]$ NuSMV wrongUpdateRule.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

file wrongUpdateRule.smv: line 6: cannot assign value 5 to variable foo

NuSMV terminated by a signal
```

NuSMV signals that it's not possible to update variable foo to value 5. In fact, since there is no control, the variable is incremented until it reaches a value that not belongs to its type.
It's important to underline that also codes like code 5.6, where it's impossible that a location assumes values not belonging to the domain, have the same problem.

Code 5.6: Update rule 2: AsmetaL model

```
asm concrDomCheck
import ./StandardLibrary

signature:
    domain MyDomain subsetof Integer
    dynamic controlled cond: Boolean
    dynamic controlled foo: MyDomain

definitions:
    domain MyDomain = {1..4}

    main  rule r_Main =
        par
            cond := not(cond)
            if(cond) then
                foo := foo + 1
            else
```

```
                foo := foo - 1
            endif
        endpar

default init s0:
    function foo = 1
    function cond = true
```

In code 5.6 location *foo* can assume only values 1 or 2; so the simulator doesn't return any error during the execution. We expect that the correct mapping should be that shown in code 5.7.

Code 5.7: Update rule 2: wrong NuSMV model

```
MODULE main
    VAR
        cond: boolean;
        foo: 1..4;
    ASSIGN
        init(cond) := TRUE;
        init(foo) := 1;
        next(cond) := !(cond);
        next(foo) :=
            case
                !(cond): (foo - 1);
                cond: (foo + 1);
                TRUE: foo;
            esac;
```

In code 5.7 variable *foo* can assume only values 1 or 2; nonetheless even the execution of code 5.7 gives an error.

```
[user@localhost tosmv]$ NuSMV concrDomCheckErr.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

file concrDomCheckErr.smv: line 14: cannot assign value 5 to variable foo

NuSMV terminated by a signal
```

To solve this problem, for each update rule of a numeric location we add a condition that states that the value of term t must be contained in domain D_l of location l:

$$t \text{ in } D_l$$

The correct translation of AsmetaL code 5.4 is shown in code 5.8; the correct translation of AsmetaL code 5.6 is shown in code 5.9.

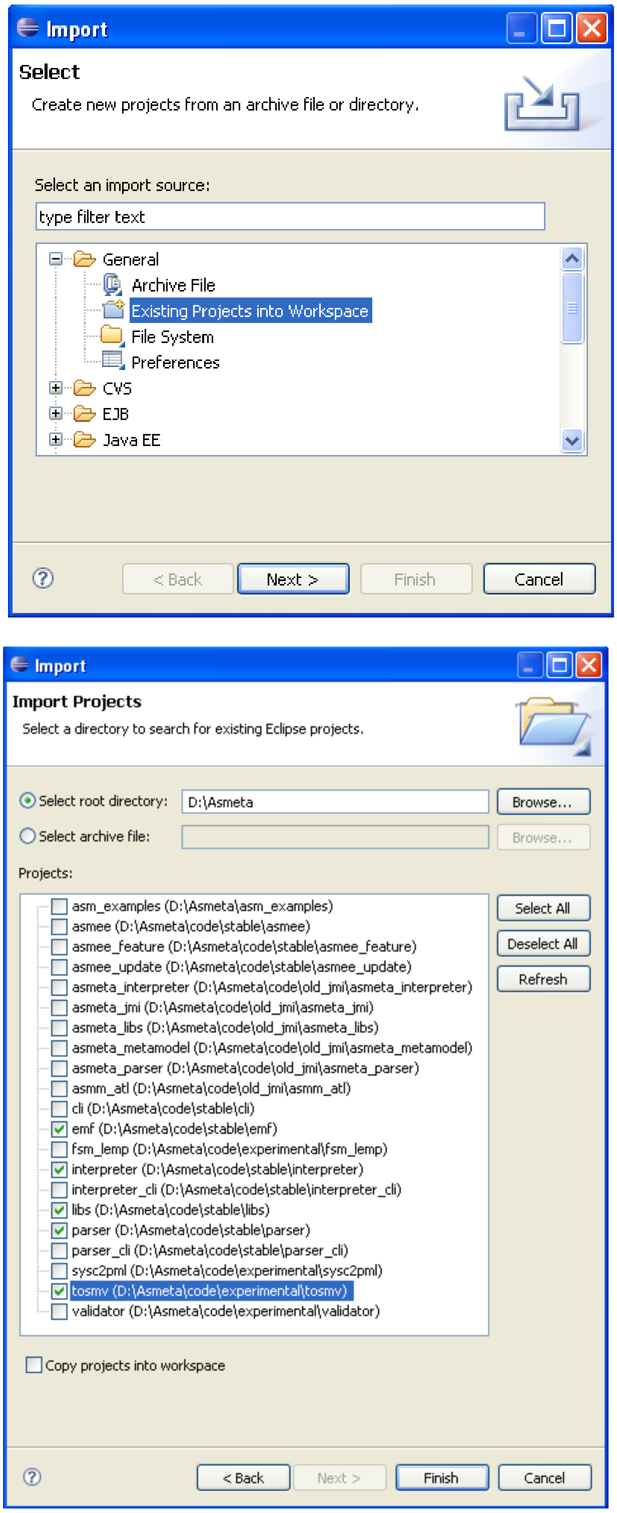
Code 5.8: Update rule 1: right NuSMV model

```
MODULE main
    VAR
        foo: 1..4;
    ASSIGN
        init(foo) := 1;
        next(foo) :=
            case
                (foo + 1) in 1..4: (foo + 1);
                TRUE: foo;
            esac;
```

Code 5.9: Update rule 2: right NuSMV model

```
MODULE main
    VAR
        cond: boolean;
        foo: 1..4;
    ASSIGN
        init(cond) := TRUE;
        init(foo) := 1;
        next(cond) := !(cond);
        next(foo) :=
            case
                !(cond) & (foo - 1) in 1..4: (foo - 1);
                cond & (foo + 1) in 1..4: (foo + 1);
                TRUE: foo;
            esac;
```

The tool, by default, adds the conditions; the execution option "-nc" permits not to add them. In fact, the adding of the conditions could change the behaviour of the model and break the equivalence between the AsmetaL model and the NuSMV one.



Import

Select

Create new projects from an archive file or directory.

Select an import source:

type filter text

- General
  - Archive File
  - Existing Projects into Workspace
  - File System
  - Preferences
- CVS
- EJB
- Java EE

Import

Import Projects

Select a directory to search for existing Eclipse projects.

● Select root directory: D:\Asmeta

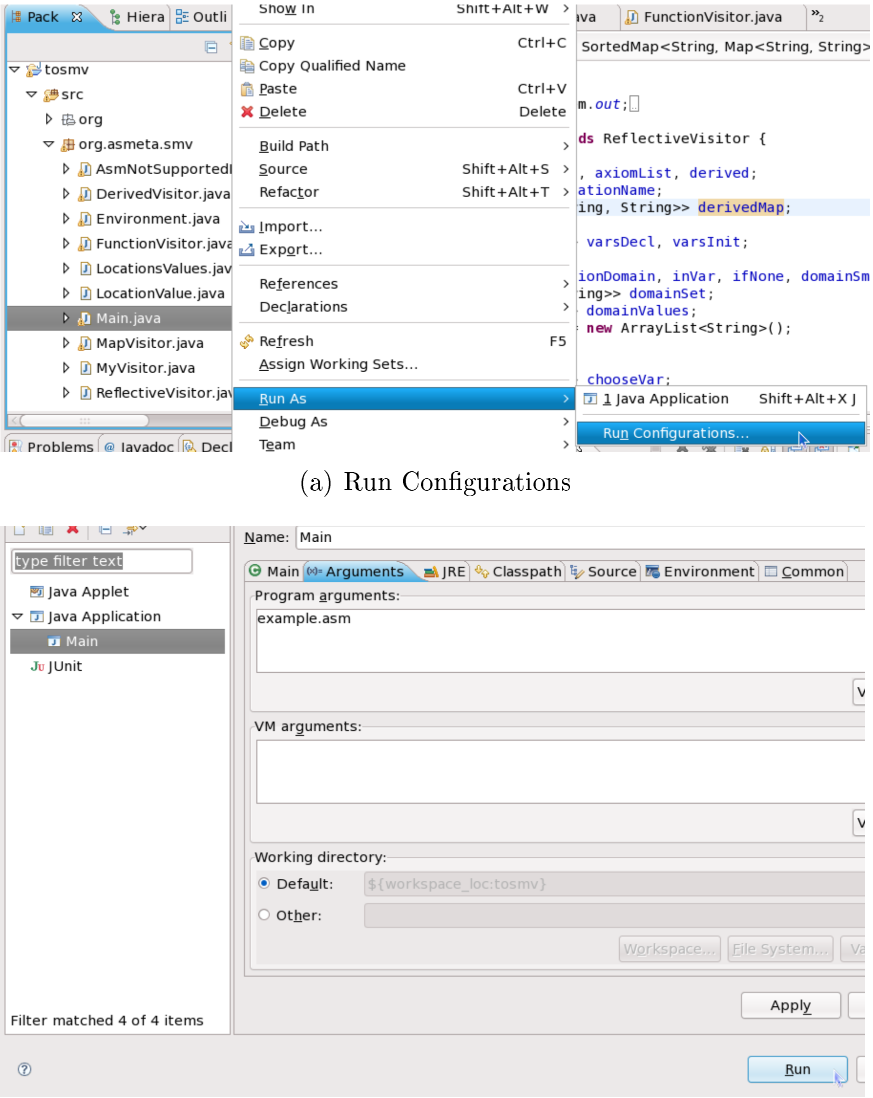
○ Select archive file:

Projects:

- □ asm_examples (D:\Asmeta\asm_examples)
- □ asmee (D:\Asmeta\code\stable\asmee)
- □ asmee_feature (D:\Asmeta\code\stable\asmee_feature)
- □ asmee_update (D:\Asmeta\code\stable\asmee_update)
- □ asmeta_interpreter (D:\Asmeta\code\old_jmi\asmeta_interpreter)
- □ asmeta_jmi (D:\Asmeta\code\old_jmi\asmeta_jmi)
- □ asmeta_libs (D:\Asmeta\code\old_jmi\asmeta_libs)
- □ asmeta_metamodel (D:\Asmeta\code\old_jmi\asmeta_metamodel)
- □ asmeta_parser (D:\Asmeta\code\old_jmi\asmeta_parser)
- □ asmm_atl (D:\Asmeta\code\old_jmi\asmm_atl)
- □ cli (D:\Asmeta\code\stable\cli)
- ☑ emf (D:\Asmeta\code\stable\emf)
- □ fsm_lemp (D:\Asmeta\code\experimental\fsm_lemp)
- ☑ interpreter (D:\Asmeta\code\stable\interpreter)
- □ interpreter_cli (D:\Asmeta\code\stable\interpreter_cli)
- ☑ libs (D:\Asmeta\code\stable\libs)
- ☑ parser (D:\Asmeta\code\stable\parser)
- □ parser_cli (D:\Asmeta\code\stable\parser_cli)
- □ sysc2pml (D:\Asmeta\code\experimental\sysc2pml)
- ☑ tosmv (D:\Asmeta\code\experimental\tosmv)
- □ validator (D:\Asmeta\code\experimental\validator)

□ Copy projects into workspace

Figure 5.1: Import of projects in Eclipse



```
SortedMap<String, Map<String, String>
m.out;
ds ReflectiveVisitor {
, axiomList, derived;
ationName;
ing, String>> derivedMap;
 varsDecl, varsInit;
ionDomain, inVar, ifNone, domainSm
ing>> domainSet;
 domainValues;
 new ArrayList<String>();
 chooseVar;
```

(a) Run Configurations

Name: Main

Program arguments: example.asm

Working directory:

● Default: ${workspace_loc:tosmv}

(b) Run

Figure 5.2: Tool execution in Eclipse

Chapter 6

Examples

In this chapter we describe some examples of use of AsmetaSMV.
We give the informal description of a problem, the ASM model written in AsmetaL and the NuSMV code obtained from the execution of the tool.
In the writing of the AsmetaL code we have considered that it had to be mapped into NuSMV; so, we didn't use elements that cannot be translated.
For each AsmetaL model we declare some properties we want to check with NuSMV.

6.1 One way traffic light control

6.1.1 Problem

In [14] it's described the specification of a system made of two traffic lights (*LIGHTUNIT1* and *LIGHTUNIT2*) placed at the beginning and at the end of an alternated one-way street; both traffic lights are connected to a computer that controls them. Each traffic light is equipped with a *Stop* light (red light) and a *Go* light (green light). The computer turns on and off the lights sending to the traffic lights two signals, *Rpulses* and *Gpulses*, that inform the traffic light to perform the switch[1], respectively, of the red and of the green light. The state of the lights of the two traffic lights changes following a four phases cycle:

- for 50 seconds both traffic lights show the *Stop* signal;
- for 120 seconds *LIGHTUNIT2* shows the *Stop* signal and *LIGHTUNIT1* the *Go* signal;

[1] to turn on/off the light if it's on/off

- for 50 seconds both traffic lights show again the *Stop* signal;
- for 120 seconds *LIGHTUNIT1* shows the *Stop* signal and *LIGHTUNIT2* the *Go* signal.

When a traffic light has the *Go* signal turned on, the cars waiting at that entry of the street can pass through. In the following period, when both units have the *Stop* signal, no cars can enter from both entries of the street; in this period the cars that are driving in one direction have the time to exit the street. In the following period, instead, the cars waiting at the other entry of the street can pass through, and so on.
Figure 6.1 shows how the system works.

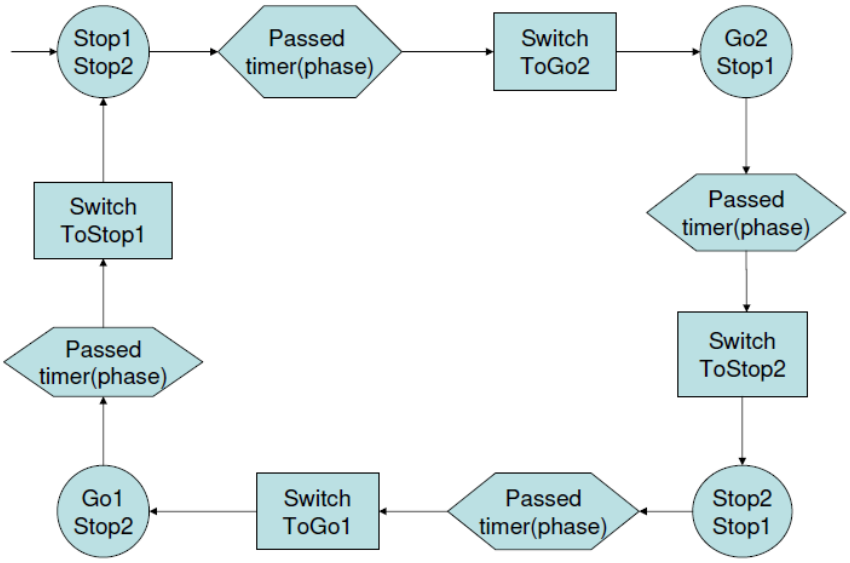


Figure 6.1: One-Way traffic light model

6.1.2 Ground model

Let's see now how the model has been written in AsmetaL; by now, we consider the ground model (code 6.1), where it's not considered the emission of *Rpulses* and *Gpulses* signals.

Code 6.1: One-way traffic light: AsmetaL ground model

```
asm oneWayTrafficLight
import ./StandardLibrary
import ./CTLlibrary
```

```
signature:
    enum domain LightUnit = {LIGHTUNIT1 | LIGHTUNIT2}
    enum domain PhaseDomain = { STOP1STOP2 | GO2STOP1 | STOP2STOP1 |
    GO1STOP2 }
    domain Intervals subsetof Integer
    dynamic controlled phase: PhaseDomain
    dynamic controlled stopLight: LightUnit -> Boolean
    dynamic controlled goLight: LightUnit -> Boolean
    dynamic monitored passed: Intervals -> Boolean

definitions:
    domain Intervals = {50, 120}

    macro rule r_switch($l in Boolean) =
        $l := not($l)

    rule r_switchLightUnit($l in LightUnit) =
        par
            r_switch[goLight($l)]
            r_switch[stopLight($l)]
        endpar

    rule r_stop1stop2_to_go2stop1 =
        if(phase=STOP1STOP2 and passed(50)) then
            par
                r_switchLightUnit[LIGHTUNIT2]
                phase:=GO2STOP1
            endpar
        endif

    rule r_go2stop1_to_stop2stop1 =
        if(phase=GO2STOP1 and passed(120)) then
            par
                r_switchLightUnit[LIGHTUNIT2]
                phase:=STOP2STOP1
            endpar
        endif

    rule r_stop2stop1_to_go1stop2 =
        if(phase=STOP2STOP1 and passed(50)) then
            par
                r_switchLightUnit[LIGHTUNIT1]
                phase:=GO1STOP2
            endpar
        endif

    rule r_go1stop2_to_stop1stop2 =
        if(phase=GO1STOP2 and passed(120)) then
            par
                r_switchLightUnit[LIGHTUNIT1]
                phase:=STOP1STOP2
            endpar
        endif

    //in each state a traffic light is red or green
    axiom over goLight: ag(goLight(LIGHTUNIT1) xor stopLight(LIGHTUNIT1))
    axiom over goLight: ag(goLight(LIGHTUNIT2) xor stopLight(LIGHTUNIT2))

    //if a traffic light is green the other one is red,
    //otherwise are both red
    axiom over goLight: ag((goLight(LIGHTUNIT2) and stopLight(LIGHTUNIT1))
    xor
                 (goLight(LIGHTUNIT1) and stopLight(LIGHTUNIT2)) xor
                 (stopLight(LIGHTUNIT2) and stopLight(LIGHTUNIT1)))

    //equal to the previous property
    axiom over goLight: ag(not(goLight(LIGHTUNIT1) and goLight(LIGHTUNIT2)))

    //properties about the right association between lights and states
    axiom over phase: ag((phase=STOP1STOP2 or phase=STOP2STOP1) iff
                  (stopLight(LIGHTUNIT1) and stopLight(LIGHTUNIT2)) )
    axiom over phase: ag(phase=GO1STOP2 iff (goLight(LIGHTUNIT1) and
    stopLight(LIGHTUNIT2)))
    axiom over phase: ag(phase=GO2STOP1 iff (goLight(LIGHTUNIT2) and
    stopLight(LIGHTUNIT1)))

    //properties about the correctness of transitions between states
    axiom over phase: ag(phase=STOP1STOP2 iff ax(phase=GO2STOP1 or phase=
    STOP1STOP2))
    axiom over phase: ag(phase=GO2STOP1 iff ax(phase=STOP2STOP1 or phase=
    GO2STOP1))
    axiom over phase: ag(phase=STOP2STOP1 iff ax(phase=GO1STOP2 or phase=
    STOP2STOP1))
    axiom over phase: ag(phase=GO1STOP2 iff ax(phase=STOP1STOP2 or phase=
    GO1STOP2))

    //if the traffic light is red, sooner or later it will be green
    axiom over stopLight: ag(stopLight(LIGHTUNIT1) implies ef(goLight(
    LIGHTUNIT1)))
    axiom over stopLight: ag(stopLight(LIGHTUNIT2) implies ef(goLight(
    LIGHTUNIT2)))

    //absence of deadlock
    axiom over phase: ag(ex(true))

    main rule r_Main =
        par
            r_stop1stop2_to_go2stop1[]
            r_go2stop1_to_stop2stop1[]
            r_stop2stop1_to_go1stop2[]
            r_go1stop2_to_stop1stop2[]
        endpar

default init s0:
    function stopLight($l in LightUnit) = true
    function goLight($l in LightUnit) = false
    function phase = STOP1STOP2
```

To represent the two traffic lights we define the enum domain *LightUnit* made of two elements, *LIGHTUNIT1* and *LIGHTUNIT2*. The stop and go lights are represented by controlled boolean functions *stopLight($l in LightUnit)* and *goLight($l in LightUnit)*; the *stopLight(LIGHTUNIT1)* location, for example, is *true* if the stop light of LIGHTUNIT1 is turned on, *false* otherwise. The system can be in four states that are listed in the enum domain *PhaseDomain*:

- *STOP1STOP2* and *STOP2STOP1*: both units show the stop light;
- *GO1STOP2*: *LIGHTUNIT1* shows the go light and *LIGHTUNIT2* the

stop light;

- *GO2STOP1*: *LIGHTUNIT2* shows the go light and *LIGHTUNIT1* the stop light.

The *phase* function, whose codomain is *PhaseDomain*, shows the current phase of the system. We also define the subset domain of Integer *Intervals* that contains values 50 and 120, that is the lengths of stop and go signals.
The location *passed($i)* of the boolean monitored function *passed($i in Intervals)* is true when the period $\$i$ is finished.
First of all we define two rules that simplify the ASM structure:

- *r_switch($l in Boolean)* switches the value of $\$l$ variable;
- *r_switchLightUnit($l in LightUnit)* switches both the lights of traffic light *$l*.

The main rule executes four rules that change the state of the system:

- *r_stop1stop2_to_go2stop1*,
- *r_go2stop1_to_stop2stop1*,
- *r_stop2stop1_to_go1stop2*,
- *r_go1stop2_to_stop1stop2*.

Let's see *r_stop1stop2_to_go2stop1* rule. If the system is in $STOP1STOP2$ phase ($phase = STOP1STOP2$) and the 50 seconds period is over ($passed(50)$), the *LIGHTUNIT2* lights switch their values with the macro call rule *r_switchLightUnit[LIGHTUNIT2]* and the system goes in state *GO2STOP1* (*phase:=GO2STOP1*).

At the beginning (default init) the system is in $STOP2STOP1$ and both units show the stop signal.

We can now describe the properties we want to check.
Safety properties

```
ag(goLight(LIGHTUNIT1) xor stopLight(LIGHTUNIT1))
ag(goLight(LIGHTUNIT2) xor stopLight(LIGHTUNIT2))
```

check that, in each unit, is turned on only one light.
Safety property

```
ag((goLight(LIGHTUNIT2) and stopLight(LIGHTUNIT1)) xor
   (goLight(LIGHTUNIT1) and stopLight(LIGHTUNIT2)) xor
   (stopLight(LIGHTUNIT2) and stopLight(LIGHTUNIT1)))
```

checks that, in each state, the combination of lights of the two units is correct. If a unit shows the go signal, the other one must show the stop signal; otherwise both units can show the stop signal.
An equivalent safety property is

```
ag(not(goLight(LIGHTUNIT1) and goLight(LIGHTUNIT2)))
```

that says that the two units can never show, at the same time, the go signal. Since now we have checked properties that must be true in all the states (temporal operator AG). Such properties are equal to AsmetaL axioms, that are properties that are checked at each step of the execution of an AsmetaL model. The translation of axioms in NuSMV permits us to verify them in all the states of the machine with one execution of the model checker; to have the same result with the simulator we should execute the ASM as many times as the number of states.
Let's see now some properties that can be verified only by NuSMV, because they verify also the future states of the machine.
Safety properties

```
ag(phase=STOP1STOP2 iff ax(phase=GO2STOP1 or phase=STOP1STOP2))
ag(phase=GO2STOP1 iff ax(phase=STOP2STOP1 or phase=GO2STOP1))
ag(phase=STOP2STOP1 iff ax(phase=GO1STOP2 or phase=STOP2STOP1))
ag(phase=GO1STOP2 iff ax(phase=STOP1STOP2 or phase=GO1STOP2))
```

check that the next state of each state is correct. For example, the first property checks that if *phase=STOP1STOP2* in the next state *phase* is $GO2STOP1$ or $STOP1STOP2$. We can notice that there are transitions where *phase* do not change; in fact, in our example, if *passed*(50) is $false$ the *phase* value remains $STOP1STOP2$.
Liveness properties

```
ag(stopLight(LIGHTUNIT1) implies ef(goLight(LIGHTUNIT1)))
ag(stopLight(LIGHTUNIT2) implies ef(goLight(LIGHTUNIT2)))
```

check that, if a traffic light shows the stop signal, sooner or later it will show the go signal.
Property

```
ag(ex(true))
```

checks the absence of deadlock; in each state there must be at least one next state.
Code 6.2 shows the NuSMV code obtained from the mapping of AsmetaL code 6.1.

Code 6.2: One-way traffic light: NuSMV ground model

```
MODULE main
    VAR
        goLight_LIGHTUNIT1: boolean;
        goLight_LIGHTUNIT2: boolean;
        passed_120: boolean;
        passed_50: boolean;
        phase: {GO1STOP2, GO2STOP1, STOP1STOP2, STOP2STOP1};
        stopLight_LIGHTUNIT1: boolean;
        stopLight_LIGHTUNIT2: boolean;
    ASSIGN
        init(goLight_LIGHTUNIT1) := FALSE;
        init(goLight_LIGHTUNIT2) := FALSE;
        init(phase) := STOP1STOP2;
        init(stopLight_LIGHTUNIT1) := TRUE;
        init(stopLight_LIGHTUNIT2) := TRUE;
        next(goLight_LIGHTUNIT1) :=
            case
                (((phase = STOP2STOP1) & (next(passed_50))) |
         ((phase = GO1STOP2) & (next(passed_120)))):
                    !(goLight_LIGHTUNIT1);
                TRUE: goLight_LIGHTUNIT1;
            esac;
        next(goLight_LIGHTUNIT2) :=
            case
                (((phase = STOP1STOP2) & (next(passed_50))) |
         ((phase = GO2STOP1) & (next(passed_120)))):
                    !(goLight_LIGHTUNIT2);
                TRUE: goLight_LIGHTUNIT2;
            esac;
        next(phase) :=
            case
                ((phase = STOP2STOP1) & (next(passed_50))): GO1STOP2;
                ((phase = STOP1STOP2) & (next(passed_50))): GO2STOP1;
                ((phase = GO2STOP1) & (next(passed_120))): STOP2STOP1;
                ((phase = GO1STOP2) & (next(passed_120))): STOP1STOP2;
                TRUE: phase;
            esac;
        next(stopLight_LIGHTUNIT1) :=
            case
                (((phase = STOP2STOP1) & (next(passed_50))) | ((phase =
    GO1STOP2) & (next(passed_120)))): !(stopLight_LIGHTUNIT1);
                TRUE: stopLight_LIGHTUNIT1;
            esac;
        next(stopLight_LIGHTUNIT2) :=
            case
                (((phase = STOP1STOP2) & (next(passed_50))) | ((phase =
    GO2STOP1) & (next(passed_120)))): !(stopLight_LIGHTUNIT2);
                TRUE: stopLight_LIGHTUNIT2;
            esac;
SPEC     AG((goLight_LIGHTUNIT1) xor (stopLight_LIGHTUNIT1));
SPEC     AG((goLight_LIGHTUNIT2) xor (stopLight_LIGHTUNIT2));
SPEC     AG((((goLight_LIGHTUNIT2) & (stopLight_LIGHTUNIT1)) xor ((
    goLight_LIGHTUNIT1) & (stopLight_LIGHTUNIT2))) xor ((
    stopLight_LIGHTUNIT2) & (stopLight_LIGHTUNIT1)));
```

```
SPEC     AG(!((goLight_LIGHTUNIT1) & (goLight_LIGHTUNIT2)));
SPEC     AG((phase = STOP1STOP2) | (phase = STOP2STOP1) <-> (
    stopLight_LIGHTUNIT1) & (stopLight_LIGHTUNIT2));
SPEC     AG(phase = GO1STOP2 <-> (goLight_LIGHTUNIT1) & (
    stopLight_LIGHTUNIT2));
SPEC     AG(phase = GO2STOP1 <-> (goLight_LIGHTUNIT2) & (
    stopLight_LIGHTUNIT1));
SPEC     AG(phase = STOP1STOP2 <-> AX((phase = GO2STOP1) | (phase =
    STOP1STOP2)));
SPEC     AG(phase = GO2STOP1 <-> AX((phase = STOP2STOP1) | (phase = GO2STOP1
    )));
SPEC     AG(phase = STOP2STOP1 <-> AX((phase = GO1STOP2) | (phase =
    STOP2STOP1)));
SPEC     AG(phase = GO1STOP2 <-> AX((phase = STOP1STOP2) | (phase = GO1STOP2
    )));
SPEC     AG(stopLight_LIGHTUNIT1 -> EF(goLight_LIGHTUNIT1));
SPEC     AG(stopLight_LIGHTUNIT2 -> EF(goLight_LIGHTUNIT2));
SPEC     AG(EX(TRUE));
```

The execution of NuSMV code verifies the correctness of the properties:

```
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

-- specification AG (goLight(LIGHTUNIT1) xor stopLight(LIGHTUNIT1))  is true
-- specification AG (goLight(LIGHTUNIT2) xor stopLight(LIGHTUNIT2))  is true
-- specification AG (((goLight(LIGHTUNIT2) & stopLight(LIGHTUNIT1)) xor
(goLight(LIGHTUNIT1) & stopLight(LIGHTUNIT2))) xor
(stopLight(LIGHTUNIT2) & stopLight(LIGHTUNIT1)))  is true
-- specification AG !(goLight(LIGHTUNIT1) & goLight(LIGHTUNIT2))  is true
-- specification AG ((phase = STOP1STOP2 | phase = STOP2STOP1) <->
(stopLight(LIGHTUNIT1) & stopLight(LIGHTUNIT2)))  is true
-- specification AG (phase = GO1STOP2 <->
(goLight(LIGHTUNIT1) & stopLight(LIGHTUNIT2)))  is true
-- specification AG (phase = GO2STOP1 <->
(goLight(LIGHTUNIT2) & stopLight(LIGHTUNIT1)))  is true
-- specification AG (phase = STOP1STOP2 <->
AX (phase = GO2STOP1 | phase = STOP1STOP2))  is true
-- specification AG (phase = GO2STOP1 <->
AX (phase = STOP2STOP1 | phase = GO2STOP1))  is true
-- specification AG (phase = STOP2STOP1 <->
AX (phase = GO1STOP2 | phase = STOP2STOP1))  is true
-- specification AG (phase = GO1STOP2 <->
AX (phase = STOP1STOP2 | phase = GO1STOP2))  is true
-- specification AG (stopLight(LIGHTUNIT1) -> EF goLight(LIGHTUNIT1))  is true
-- specification AG (stopLight(LIGHTUNIT2) -> EF goLight(LIGHTUNIT2))  is true
-- specification AG (EX TRUE)  is true
```

6.1.3 Refined model

Let's see now the refined model, where we introduce *Rpulses* and *Gpulses* signals (code 6.3).

Code 6.3: One-way traffic light: AsmetaL refined model

```
asm oneWayTrafficLight_refined
import ./StandardLibrary
import ./CTLlibrary

signature:
    enum domain LightUnit = {LIGHTUNIT1 | LIGHTUNIT2}
    enum domain PhaseDomain = { STOP1STOP2 | GO2STOP1 | STOP2STOP1
    | GO1STOP2 | STOP1STOP2CHANGING | GO2STOP1CHANGING
    | STOP2STOP1CHANGING | GO1STOP2CHANGING }
    domain Intervals subsetof Integer
    dynamic controlled phase: PhaseDomain
    dynamic controlled stopLight: LightUnit -> Boolean
    dynamic controlled goLight: LightUnit -> Boolean
    dynamic monitored passed: Intervals -> Boolean
    dynamic controlled rPulse: LightUnit -> Boolean
    dynamic controlled gPulse: LightUnit -> Boolean

definitions:
    domain Intervals = {50, 120}

    macro rule r_switch($l in Boolean) =
        $l := not($l)

    rule r_switchLightUnit($l in LightUnit) =
        par
            rPulse($l) := true
            gPulse($l) := true
        endpar

    rule r_stop1stop2_to_stop1stop2changing =
        if(phase=STOP1STOP2 and passed(50)) then
            par
                r_switchLightUnit[LIGHTUNIT2]
                phase:=STOP1STOP2CHANGING
            endpar
        endif

    rule r_go2stop1_to_go2stop1changing =
        if(phase=GO2STOP1 and passed(120)) then
            par
                r_switchLightUnit[LIGHTUNIT2]
                phase:=GO2STOP1CHANGING
            endpar
        endif

    rule r_stop2stop1_to_stop2stop1changing =
        if(phase=STOP2STOP1 and passed(50)) then
            par
                r_switchLightUnit[LIGHTUNIT1]
                phase:=STOP2STOP1CHANGING
            endpar
        endif

    rule r_go1stop2_to_go1stop2changing =
        if(phase=GO1STOP2 and passed(120)) then
```

```
            par
                r_switchLightUnit[LIGHTUNIT1]
                phase:=GO1STOP2CHANGING
            endpar
        endif

    rule r_pulses =
        forall $l in LightUnit with true do
            par
                if(gPulse($l)) then
                    par
                        r_switch[goLight($l)]
                        gPulse($l) := false
                    endpar
                endif
                if(rPulse($l)) then
                    par
                        r_switch[stopLight($l)]
                        rPulse($l) := false
                    endpar
                endif
            endpar

    macro rule r_changeState =
        par
            if(phase=STOP1STOP2CHANGING) then
                phase := GO2STOP1
            endif
            if(phase=GO2STOP1CHANGING) then
                phase := STOP2STOP1
            endif
            if(phase=STOP2STOP1CHANGING) then
                phase := GO1STOP2
            endif
            if(phase=GO1STOP2CHANGING) then
                phase := STOP1STOP2
            endif
        endpar

    //rPulse and gPulse signals are read and cleared in one step
    axiom over rPulse: ag(rPulse(LIGHTUNIT1) implies ax(not(rPulse(
    LIGHTUNIT1))))
    axiom over rPulse: ag(rPulse(LIGHTUNIT2) implies ax(not(rPulse(
    LIGHTUNIT2))))
    axiom over gPulse: ag(gPulse(LIGHTUNIT1) implies ax(not(gPulse(
    LIGHTUNIT1))))
    axiom over gPulse: ag(gPulse(LIGHTUNIT2) implies ax(not(gPulse(
    LIGHTUNIT2))))

    main rule r_Main =
        par
            r_stop1stop2_to_stop1stop2changing[]
            r_go2stop1_to_go2stop1changing[]
            r_stop2stop1_to_stop2stop1changing[]
            r_go1stop2_to_go1stop2changing[]
            r_pulses[]
            r_changeState[]
        endpar

default init s0:
    function stopLight($l in LightUnit) = true
    function goLight($l in LightUnit) = false
    function phase = STOP1STOP2
    function rPulse($l in LightUnit) = false
    function gPulse($l in LightUnit) = false
```

Rpulses and *Gpulses* are represented with two boolean controlled functions, *rPulse($l LightUnit)* and *gPulse($l LightUnit)*. For example, location $rPulse(LIGHTUNIT1)$ is *true* if the computer has sent to $LIGHTUNIT1$ the signal to turn on the red light.
Besides the four state previously described, we define four more states:

- *STOP1STOP2CHANGING*: both traffic lights show the stop light; *LIGHTUNIT2* has received from the computer the signal to turn on the go light and to turn off the stop light;

- *GO2STOP1CHANGING*: $LIGHTUNIT2$ shows the go light and *LIGHTUNIT1* the stop light; $LIGHTUNIT2$ has received from the computer the signal to turn off the go light and to turn on the stop light;

- *STOP2STOP1CHANGING*: both traffic lights show the stop light; $LIGHTUNIT1$ has received from the computer the signal to turn on the go light and to turn off the stop light;

- *GO1STOP2CHANGING*: $LIGHTUNIT1$ shows the go light and *LIGHTUNIT2* the stop light; $LIGHTUNIT1$ has received from the computer the signal to turn off the go light and to turn on the stop light.

Rule *r_switchLightUnit($l in LightUnit)* has been modified: it doesn't execute anymore the switch of the lights of traffic light *$l*, but sends to it two switch signals: it sets to true the locations *rPulse($l)* and *gPulse($l)*.
The rules that, in the ground model, execute the transition between the main states, have been replaced by other four rules that execute the transitions between the main states and the intermediate states. The new rules are:

- *r_stop1stop2_to_stop1stop2changing*,

- *r_go2stop1_to_go2stop1changing*,

- *r_stop2stop1_to_stop2stop1changing*,

- *r_go1stop2_to_go1stop2changing*.

These four rules, with the rules *r_pulses* and *r_changeState*, are executed in the main rule.
r_pulses reads *rPulse($l LightUnit)* and *gPulse($l LightUnit)* signals and turn on/off the lights of the traffic lights.

r_ changeState executes the transitions between the intermediate states and the principal states.
We declare four safety properties to check the communication between the computer and the traffic lights; signals, when are received by the traffic lights, are immediately read and cleared (updated to *false*).

```
ag(rPulse(LIGHTUNIT1) implies ax(not(rPulse(LIGHTUNIT1))))
ag(rPulse(LIGHTUNIT2) implies ax(not(rPulse(LIGHTUNIT2))))
ag(gPulse(LIGHTUNIT1) implies ax(not(gPulse(LIGHTUNIT1))))
ag(gPulse(LIGHTUNIT2) implies ax(not(gPulse(LIGHTUNIT2))))
```

Code 6.4 is the translation into NuSMV of AsmetaL code 6.3.

Code 6.4: One-way traffic light: NuSMV refined model

```
MODULE main
    VAR
        gPulse_LIGHTUNIT1: boolean;
        gPulse_LIGHTUNIT2: boolean;
        goLight_LIGHTUNIT1: boolean;
        goLight_LIGHTUNIT2: boolean;
        passed_120: boolean;
        passed_50: boolean;
        phase: {GO1STOP2, GO1STOP2CHANGING, GO2STOP1, GO2STOP1CHANGING,
    STOP1STOP2, STOP1STOP2CHANGING, STOP2STOP1, STOP2STOP1CHANGING};
        rPulse_LIGHTUNIT1: boolean;
        rPulse_LIGHTUNIT2: boolean;
        stopLight_LIGHTUNIT1: boolean;
        stopLight_LIGHTUNIT2: boolean;
    ASSIGN
        init(gPulse_LIGHTUNIT1) := FALSE;
        init(gPulse_LIGHTUNIT2) := FALSE;
        init(goLight_LIGHTUNIT1) := FALSE;
        init(goLight_LIGHTUNIT2) := FALSE;
        init(phase) := STOP1STOP2;
        init(rPulse_LIGHTUNIT1) := FALSE;
        init(rPulse_LIGHTUNIT2) := FALSE;
        init(stopLight_LIGHTUNIT1) := TRUE;
        init(stopLight_LIGHTUNIT2) := TRUE;
        next(gPulse_LIGHTUNIT1) :=
            case
                (gPulse_LIGHTUNIT1): FALSE;
                (((phase = STOP2STOP1) & (next(passed_50))) |
         ((phase = GO1STOP2) & (next(passed_120)))): TRUE;
                TRUE: gPulse_LIGHTUNIT1;
            esac;
        next(gPulse_LIGHTUNIT2) :=
            case
                (((phase = STOP1STOP2) & (next(passed_50))) |
         ((phase = GO2STOP1) & (next(passed_120)))): TRUE;
                (gPulse_LIGHTUNIT2): FALSE;
                TRUE: gPulse_LIGHTUNIT2;
            esac;
        next(goLight_LIGHTUNIT1) :=
            case
                (gPulse_LIGHTUNIT1): !(goLight_LIGHTUNIT1);
                TRUE: goLight_LIGHTUNIT1;
            esac;
        next(goLight_LIGHTUNIT2) :=
            case
                (gPulse_LIGHTUNIT2): !(goLight_LIGHTUNIT2);
                TRUE: goLight_LIGHTUNIT2;
            esac;
        next(phase) :=
            case
                (phase = GO1STOP2CHANGING): STOP1STOP2;
                ((phase = STOP2STOP1) & (next(passed_50))):
    STOP2STOP1CHANGING;
                (phase = STOP1STOP2CHANGING): GO2STOP1;
                (phase = STOP2STOP1CHANGING): GO1STOP2;
                ((phase = STOP1STOP2) & (next(passed_50))):
    STOP1STOP2CHANGING;
                (phase = GO2STOP1CHANGING): STOP2STOP1;
                ((phase = GO2STOP1) & (next(passed_120))): GO2STOP1CHANGING;
                ((phase = GO1STOP2) & (next(passed_120))): GO1STOP2CHANGING;
                TRUE: phase;
            esac;
        next(rPulse_LIGHTUNIT1) :=
            case
                (rPulse_LIGHTUNIT1): FALSE;
                (((phase = STOP2STOP1) & (next(passed_50))) | ((phase =
    GO1STOP2) & (next(passed_120)))): TRUE;
                TRUE: rPulse_LIGHTUNIT1;
            esac;
        next(rPulse_LIGHTUNIT2) :=
            case
                (rPulse_LIGHTUNIT2): FALSE;
                (((phase = STOP1STOP2) & (next(passed_50))) | ((phase =
    GO2STOP1) & (next(passed_120)))): TRUE;
                TRUE: rPulse_LIGHTUNIT2;
            esac;
        next(stopLight_LIGHTUNIT1) :=
            case
                (rPulse_LIGHTUNIT1): !(stopLight_LIGHTUNIT1);
                TRUE: stopLight_LIGHTUNIT1;
            esac;
        next(stopLight_LIGHTUNIT2) :=
            case
                (rPulse_LIGHTUNIT2): !(stopLight_LIGHTUNIT2);
                TRUE: stopLight_LIGHTUNIT2;
            esac;
SPEC     AG(rPulse_LIGHTUNIT1 -> AX(!(rPulse_LIGHTUNIT1)));
SPEC     AG(rPulse_LIGHTUNIT2 -> AX(!(rPulse_LIGHTUNIT2)));
SPEC     AG(gPulse_LIGHTUNIT1 -> AX(!(gPulse_LIGHTUNIT1)));
SPEC     AG(gPulse_LIGHTUNIT2 -> AX(!(gPulse_LIGHTUNIT2)));
```

Let's check the correctness of the properties through the execution of the NuSMV code:

```
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

-- specification AG (rPulse(LIGHTUNIT1) -> AX !rPulse(LIGHTUNIT1))  is true
-- specification AG (rPulse(LIGHTUNIT2) -> AX !rPulse(LIGHTUNIT2))  is true
-- specification AG (gPulse(LIGHTUNIT1) -> AX !gPulse(LIGHTUNIT1))  is true
-- specification AG (gPulse(LIGHTUNIT2) -> AX !gPulse(LIGHTUNIT2))  is true
```

6.2 Sluice Gate Control

6.2.1 Problem

In [14] it's described the specification of an irrigation system. The irrigation system is composed by a small sluice, with a rising and a falling gate, and a computer system that controls the sluice gate. This is the description of the requirements as reported in [14]:

> The sluice gate must be held in the fully open position for ten minutes in ever three hours and otherwise kept in the fully closed position. The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensor at the top and bottom of the gate travel; at the top it's fully open, at the bottom it's fully shut.
> The connection to the computer consists of four pulse lines for motor control and two status lines for gate sensor.

6.2.2 Ground model

In the ground model the sluice gate can be only in two states: fully open or fully closed. By now we don't consider the movements of the gate from one state to the other.
We don't even consider the motor; we just model the elapsing of intervals in which the sluice gate must be opened or closed. Code 6.5 is the AsmetaL ground model.

Code 6.5: Sluice Gate Control: AsmetaL ground model

```
asm sluiceGateGround
import ./StandardLibrary
import ./CTLlibrary

signature:
    domain Minutes subsetof Integer
    enum domain PhaseDomain = {FULLYCLOSED | FULLYOPENED}
    dynamic controlled phase: PhaseDomain
    dynamic monitored passed: Minutes -> Boolean

definitions:
    domain Minutes = {10, 170}

    rule r_open =
        skip
    rule r_shut =
        skip

    //transitions between states are correct
    axiom over phase: ag(phase=FULLYCLOSED implies ax(phase = FULLYOPENED
    iff passed(170)))
    axiom over phase: ag(phase=FULLYOPENED implies ax(phase = FULLYCLOSED
    iff passed(10)))

    main rule r_Main =
        par
            if(phase=FULLYCLOSED) then
                if(passed(170)) then
                    par
                        r_open[]
                        phase := FULLYOPENED
                    endpar
                endif
            endif
            if(phase=FULLYOPENED) then
                if(passed(10)) then
                    par
                        r_shut[]
                        phase := FULLYCLOSED
                    endpar
                endif
            endif
        endpar

default init s0:
    function phase = FULLYCLOSED
```

In the AsmetaL model we have declared a function *phase* that records the state of the gate: the gate can be fully opened ($FULLYOPENED$) or fully closed ($FULLYCLOSED$). The boolean monitored locations $passed(10)$ and $passed(170)$ say if the intervals in which the gate is, respectively, fully opened and fully closed are elapsed.
The main rule simply simulates the changes of the gate state:

- if the state is $FULLYCLOSED$ and 170 minutes are elapsed, the state becomes $FULLYOPENED$;

- if the state is $FULLYOPENED$ and 10 minutes are elapsed, the state becomes $FULLYCLOSED$.

At the beginning the gate is closed.
Figure 6.2 shows how the ground model works.

In the model we have defined two safety properties

```
ag(phase=FULLYCLOSED implies ax(phase = FULLYOPENED iff passed(170)))
ag(phase=FULLYOPENED implies ax(phase = FULLYCLOSED iff passed(10)))
```

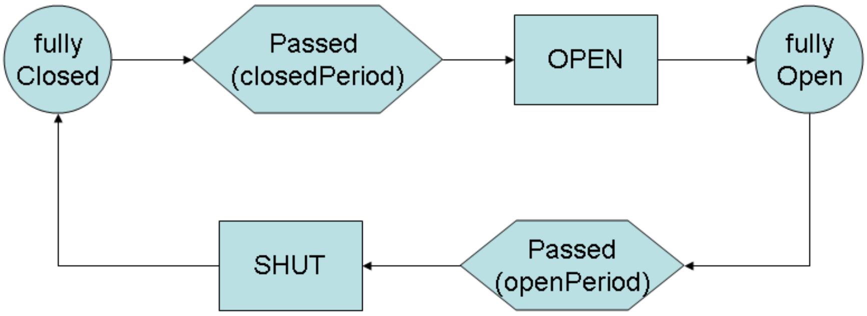


Figure 6.2: Sluice Gate Control - ground model

that verify that the transitions between states are correct. For example, if the gate is $FULLYCLOSED$, the next state of the gate will be $FULLYOPENED$ only if 170 minutes are elapsed ($passed(170)$).
Code 6.6 is the NuSMV code obtained from the mapping of AsmetaL code 6.5.

Code 6.6: Sluice Gate Control: NuSMV ground model

```
MODULE main
    VAR
        passed_10: boolean;
        passed_170: boolean;
        phase: {FULLYCLOSED, FULLYOPENED};
    ASSIGN
        init(phase) := FULLYCLOSED;
        next(phase) :=
            case
                (phase = FULLYCLOSED) & (next(passed_170)): FULLYOPENED;
                (phase = FULLYOPENED) & (next(passed_10)): FULLYCLOSED;
                TRUE: phase;
            esac;
SPEC     AG((phase = FULLYCLOSED) -> AX(phase = FULLYOPENED <-> passed_170))
    ;
SPEC     AG((phase = FULLYOPENED) -> AX(phase = FULLYCLOSED <-> passed_10));
```

We can see that, each AsmetaL location has been transformed into a NuSMV variable.
Let's execute the NuSMV code to check the properties.

```
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG (phase = FULLYCLOSED -> AX (phase = FULLYOPENED <->
passed(170)))  is true
-- specification AG (phase = FULLYOPENED -> AX (phase = FULLYCLOSED <->
passed(10)))  is true
```

6.2.3 Refined model

Let's see in this section the refined model (code 6.7).

Code 6.7: Sluice Gate Control: AsmetaL refined model

```
asm sluiceGateMotorCtl
import ./StandardLibrary
import ./CTLlibrary

signature:
    domain Minutes subsetof Integer
    enum domain PhaseDomain = { FULLYCLOSED | OPENING | FULLYOPENED |
    CLOSING }
    enum domain DirectionDomain = { CLOCKWISE | ANTICLOCKWISE }
    enum domain MotorDomain = { ON | OFF }
    dynamic controlled phase: PhaseDomain
    dynamic controlled dir: DirectionDomain
    dynamic controlled motor: MotorDomain
    dynamic monitored passed: Minutes -> Boolean
    dynamic monitored event_top: Boolean
    dynamic monitored event_bottom: Boolean

definitions:
    domain Minutes = {10, 170}

    rule r_start_to_raise =
        par
            dir := CLOCKWISE
            motor := ON
        endpar

    rule r_start_to_lower =
        par
            dir := ANTICLOCKWISE
            motor := ON
        endpar

    rule r_stop_motor =
        motor := OFF

    //correctness of phase changes
    axiom over phase: ag(phase=FULLYCLOSED implies ax(phase=FULLYCLOSED or
    phase=OPENING))
    axiom over phase: ag(phase=OPENING implies ax(phase=OPENING or phase=
    FULLYOPENED))
    axiom over phase: ag(phase=FULLYOPENED implies ax(phase=FULLYOPENED or
    phase=CLOSING))
    axiom over phase: ag(phase=CLOSING implies ax(phase=CLOSING or phase=
    FULLYCLOSED))

    //properties about the connection between the state
    //and the motor
    axiom over phase: ag(phase=FULLYCLOSED implies motor = OFF)
    axiom over phase: ag(phase=FULLYOPENED implies motor = OFF)
    axiom over phase: ag(phase=OPENING implies motor = ON)
    axiom over phase: ag(phase=CLOSING implies motor = ON)

    //liveness properties
    axiom over phase: ag(phase = FULLYOPENED implies ef(phase = FULLYCLOSED)
    )
    axiom over phase: ag(phase = FULLYCLOSED implies ef(phase = FULLYOPENED)
    )

    main rule r_Main =
        par
            if(phase=FULLYCLOSED) then
                if(passed(170)) then
                    par
                        r_start_to_raise[]
                        phase := OPENING
                    endpar
                endif
            endif
            if(phase=OPENING) then
                if(event_top) then
                    par
                        r_stop_motor[]
                        phase := FULLYOPENED
                    endpar
                endif
            endif
            if(phase=FULLYOPENED) then
                if(passed(10)) then
                    par
                        r_start_to_lower[]
                        phase := CLOSING
                    endpar
                endif
            endif
            if(phase=CLOSING) then
                if(event_bottom) then
                    par
                        r_stop_motor[]
                        phase := FULLYCLOSED
                    endpar
                endif
            endif
        endpar

default init s0:
    function phase = FULLYCLOSED
    function motor = OFF
```

In the refined model we have introduced the representation of the motor and of the opening and closing movements of the gate.
Variable *phase* can be in two new states: $OPENING$ to signal that the gate is opening, $CLOSING$ to signal that the gate is closing.
To model the motor we have introduced the variable *motor* whose value is ON if the motor is turned on, OFF if it's turned off. Variable *dir* represents the direction of the screws, $CLOCKWISE$ or $ANTICLOCKWISE$. Monitored functions *event_top* and *event_bottom* says if the gate has reached, respectively, the highest and the lowest point.
The functioning of the system follows a four phases cycle:

- if the gate is fully closed ($FULLYCLOSED$) and 170 minutes have elapsed (*passed*(170)), the computer turns on the motor and sets the screws direction to *CLOCKWISE*; the gate enters state *OPENING*;

- if the gate is opening (*OPENING*) and has reached the highest position (*event_top*), the computer turns off the motor; the gate enters state *FULLYOPENED*;
- if the gate is fully opened (*FULLYOPENED*) and 10 minutes have elapsed (*passed*(10)), the computer turns on the motor and sets the screws direction to *ANTICLOCKWISE*; the gate enters state *CLOSING*;
- if the gate is closing (*CLOSING*) and has reached the lower position (*event_bottom*), the computer turns off the motor; the gate enters state *FULLYCLOSED*.

In the model we have declared some properties.
The four safety properties

```
ag(phase=FULLYCLOSED implies ax(phase=FULLYCLOSED or phase=OPENING))
ag(phase=OPENING implies ax(phase=OPENING or phase=FULLYOPENED))
ag(phase=FULLYOPENED implies ax(phase=FULLYOPENED or phase=CLOSING))
ag(phase=CLOSING implies ax(phase=CLOSING or phase=FULLYCLOSED))
```

check that transitions between states are correct; for example, if the gate is *FULLYCLOSED*, in the next state can remain in *FULLYCLOSED* or enter in *OPENING*.
The four safety properties

```
ag(phase=FULLYCLOSED implies motor = OFF)
ag(phase=FULLYOPENED implies motor = OFF)
ag(phase=OPENING implies motor = ON)
ag(phase=CLOSING implies motor = ON)
```

check that the motor is turned off when the gate is stopped, and that is turned on when the gate is moving.
The two liveness properties

```
ag(phase = FULLYOPENED implies ef(phase = FULLYCLOSED))
ag(phase = FULLYCLOSED implies ef(phase = FULLYOPENED))
```

check that, if the gate is *FULLYCLOSED*, sooner or later it will become *FULLYOPENED* and vice versa.
Code 6.8 contains the NuSMV code obtained from the mapping of AsmetaL code 6.7.

Code 6.8: Sluice Gate Control: NuSMV refined model

```
MODULE main
    VAR
        dir: {ANTICLOCKWISE, CLOCKWISE};
        event_bottom: boolean;
        event_top: boolean;
        motor: {OFF, ON};
        passed_10: boolean;
        passed_170: boolean;
        phase: {CLOSING,FULLYCLOSED,FULLYOPENED,OPENING};
    ASSIGN
        init(motor) := OFF;
        init(phase) := FULLYCLOSED;
        next(dir) :=
            case
                (phase = FULLYCLOSED) & (next(passed_170)):
                    CLOCKWISE;
                (phase = FULLYOPENED) & (next(passed_10)):
                    ANTICLOCKWISE;
                TRUE: dir;
            esac;
        next(motor) :=
            case
                (phase = OPENING) & (next(event_top)): OFF;
                (phase = FULLYCLOSED) & (next(passed_170)): ON;
                (phase = CLOSING) & (next(event_bottom)): OFF;
                (phase = FULLYOPENED) & (next(passed_10)): ON;
                TRUE: motor;
            esac;
        next(phase) :=
            case
                (phase = OPENING) & (next(event_top)):
                    FULLYOPENED;
                (phase = FULLYCLOSED) & (next(passed_170)):
                    OPENING;
                (phase = CLOSING) & (next(event_bottom)):
                    FULLYCLOSED;
                (phase = FULLYOPENED) & (next(passed_10)):
                    CLOSING;
                TRUE: phase;
            esac;
SPEC    AG((phase = FULLYCLOSED) -> AX((phase = FULLYCLOSED) | (phase =
    OPENING)));
SPEC    AG((phase = OPENING) -> AX((phase = OPENING) | (phase = FULLYOPENED)
    ));
SPEC    AG((phase = FULLYOPENED) -> AX((phase = FULLYOPENED) | (phase =
    CLOSING)));
SPEC    AG((phase = CLOSING) -> AX((phase = CLOSING) | (phase = FULLYCLOSED)
    ));
SPEC    AG((phase = FULLYCLOSED) -> (motor = OFF));
SPEC    AG((phase = FULLYOPENED) -> (motor = OFF));
SPEC    AG((phase = OPENING) -> (motor = ON));
SPEC    AG((phase = CLOSING) -> (motor = ON));
SPEC    AG(phase = FULLYCLOSED -> EF(phase = FULLYOPENED));
SPEC    AG(phase = FULLYOPENED -> EF(phase = FULLYCLOSED));
```

Let's execute code 6.8 in order to check the correctness of the properties.

```
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG (phase = FULLYCLOSED -> AX (phase = FULLYCLOSED |
phase = OPENING))  is true
-- specification AG (phase = OPENING -> AX (phase = OPENING |
phase = FULLYOPEN))  is true
-- specification AG (phase = FULLYOPEN -> AX (phase = FULLYOPEN |
phase = CLOSING))  is true
-- specification AG (phase = CLOSING -> AX (phase = CLOSING |
phase = FULLYCLOSED))  is true
-- specification AG (phase = FULLYCLOSED -> motor = OFF)  is true
-- specification AG (phase = FULLYOPEN -> motor = OFF)  is true
-- specification AG (phase = OPENING -> motor = ON)  is true
-- specification AG (phase = CLOSING -> motor = ON)  is true
```

6.3 Mondex protocol

In [15] it's described an ASM model for the Mondex protocol [16]. The mondex protocol implements electronic cash transfer between two *purses* (*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 analyzed the first refinement described in chapter 3 ("From Atomic Transfers to Messages") of [15]. In section 6.3.1 we describe the AsmetaL model we have written; the code implements a simplified version of the ASM model described in [15]. Moreover, in writing the AsmetaL code, we have considered that it had to be translated into NuSMV and so we have used only elements that are supported by the mapping. The model contained in section 6.3.1 contains an error; in sections 6.3.2 and 6.3.3 we will see two possible solutions.

6.3.1 Model with error

We consider a simplified version of the problem:

- there are only two cards (*AA* and *BB* belonging to the domain *Name*);
- it's not possible that a message is lost, that is rule LOSEMSG of [15] is not considered;
- it's not possible that a card aborts a transition, that is rule ABORT of [15] is not considered; we will introduce the ABORT in section 6.3.2.

Signature The function *balance($n in Name)* represents the balance of card *$n*; the boolean function *authentic($n in Name)* says if card $*n* is authentic.
Each card has an inbox that contains messages awaiting processing. The boolean function *inbox($n in Name, $m in MessageType, $na in Name, $value in MoneyDomain, $t in TidDomain)* models the inboxes of the cards; the arguments are:

- *$n*: the owner of the inbox;
- *$m*: the type of the message;
- *$na*: the sender of the message;
- *$value*: the amount of money involved in the transfer; the values that can be used are the elements of *MoneyDomain* domain;
- *$t*: the identifier of the transaction; the values that can be used are the elements of *TidDomain* domain.

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 functions that contain the elements of the message:

- *outboxMessage: Name* → *MessageType*: type of the message;
- *outboxName: Name* → *Name*: the addressee of the message;
- *outboxMoney: Name* → *MoneyDomain*: the amount of money involved in the transfer;
- *outboxTid: Name* → *TidDomain*: the identifier of the transaction;
- *outboxIsNone: Name* → *Boolean*: says if the outbox contains a message.

The element of the domain *Name* of the five functions is the owner of the outbox. Let's see an example to visualize the correspondence between the inbox of a card that has received a message and the outbox of the card that has sent the same message. In this example BB has sent a REQ message of 0 money to AA; the identifier of the transaction is 1.

inbox of AA	**outbox of BB**
inbox(AA, REQ, BB, 0n, 1n) = true	outboxMessage(BB) = REQ outboxName(BB) = AA outboxMoney(BB) = 0n outboxTid(BB) = 1n outboxIsNone(BB) = false

The derived function *isNone($n Name)* says if the outbox of *$n* is not involved in a transaction: the outbox contains no message (*outboxIsNone($n) = true*) or contains an ACK message (*outboxMessage($n) = ACK*).
The boolean function *tid($t in TidDomain)* says if a tid has already been used.

Rules A transfer of money from card *card1* to card *card2* is done through the execution of this four rules:

1. *r_startTo($receiver in Name)*: *card2* requests *v* money to *card1* (it sends a REQ message in the *inbox* of *card1*) and memorizes the message in its outbox; *card2* can send the request only if it's not involved in a transaction (*isNone(card2) = true*);

2. *r_req($receiver in Name)*: *card1* receives the request, removes *v* money from its balance, removes the message from its inbox, sends a VAL message in the *inbox* of *card2* and puts the message also in its outbox; *card1* can receive the request only if it's not involved in a transaction (*isNone(card1) = true*);

3. *r_val($receiver in Name)*: *card2* receives the VAL message, adds *v* money on its balance, removes the message from its inbox, sends an ACK message in the inbox of *card1* and puts the message also in its outbox;

4. *r_ack($receiver in Name)*: *card1* receives the ACK message, removes the message from its inbox and clears its outbox.

In the main rule, nondeterministically a card and a rule are chosen; the chosen card executes the chosen rule as a receiver of the message[2].

[2] If the chosen rule is *r_startTo* the card can start a request of money

Model error We 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 *r_startTo($receiver in Name)* is the following:

1. the rule can be executed only if card *$receiver* is not involved in a previous transaction, that is *isNone($receiver)* = *true*;
2. a message *($na, REQ, $value, $tid)* is chosen such that card *$na* is authentic and different from card *$receiver*, and the tid *$tid* has not yet been used; if one of these conditions cannot be satisfied the following points can't be executed;
3. the card *$receiver* puts the message in its outbox and in the inbox of card *$na*;
4. the tid *$tid* is removed from the available tids.

The execution of *r_req($receiver in Name)* is the following:

1. the rule can be executed only if card *$receiver* is not involved in a previous transaction, that is *isNone($receiver)* = *true*;
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 *$value* is less or equal to the balance of the receiver; if one of these conditions cannot be satisfied the following points can't be executed;
3. the card *$receiver* puts a VAL message in its outbox and in the inbox of card *$na*;
4. the chosen message is removed from the inbox of the card *$receiver*;
5. the value *$value* is removed from the balance of the card *$receiver.*

Thanks to the verification of some properties with the NuSMV model, we have discovered that there is a situation in which the system is in deadlock:

1. card *BB* executes the *r_startTo rule*: it asks 0 money to card *AA*[3]. The outbox of *BB*, at the end of the rule, contains the same message it has sent to *AA*.

[3] We can notice that there will always be enough money on the balance of *AA* to satisfy a request of 0 money.

2. card *AA* executes the *r_startTo* rule: it asks 0 money to card *BB*. The outbox of *AA*, at the end of the rule, contains the same message it has sent to *BB*.

At this point, in order to continue the transfers of money, the two cards would execute the *r_req* rule to satisfy the request of the other card. The *r_req* rule, however, satisfies a REQ message only if the outbox of the receiver is empty. Unfortunately both cards have their outboxes occupied by the REQ message that they have sent to the other card. So we see that the two cards are blocked.
Three properties help us to visualize the problem.
Property

```
ag(inbox(AA, REQ, BB, 0n, 1n) implies
ef(inbox(BB, VAL, AA, 0n, 1n)))
```

check that if *BB* has done a request of 0 money to *AA*, sooner or later *AA* will reply with the VAL message. We will see that this property is false and we will be able to observe a counterexample.
The next two properties have been suggested by the counterexample returned by the previous property.
Property

```
not(ef(inbox(AA, REQ, BB, 0n, 1n) and inbox(BB, REQ, AA, 0n, 2n)))
```

verifies that exists a state in which both cards have requested 0 money from the other card. We can observe that the two requests, correctly, have two different tids. The property is negated in order to obtain a counterexample. Since both cards request 0 money, the balances of the cards are enough to satisfy the requests. Nonetheless the property

```
ag((inbox(AA,REQ,BB,0n,1n) and inbox(BB,REQ,AA,0n,2n)) implies
ag(not(inbox(BB,VAL,AA,0n, 1n) or inbox(AA,VAL,BB,0n,2n))))
```

verifies that, if the system is in the state previously described, none of the two requests can be satisfied in the future. As we have say previously, in fact, the two cards are in deadlock.
Code 6.9 is the AsmetaL code.

Code 6.9: Mondex protocol with error: AsmetaL model

```
asm mcCap3ForNuSMVoutboxSing
import ./StandardLibrary

signature:
    enum domain Name = {AA | BB} //2 cards
    enum domain MessageType = {REQ | VAL | ACK}
    enum domain RuleId = {STARTTORULE | REQRULE | VALRULE | ACKRULE}
    domain TidDomain subsetof Natural //domain for tids
    domain MoneyDomain subsetof Natural //domain for money

    dynamic controlled balance: Name -> MoneyDomain
    dynamic controlled tids: TidDomain -> Boolean //true if a tid has been
    used
    dynamic controlled inbox: Prod(Name, MessageType, Name, MoneyDomain,
    TidDomain) -> Boolean
    dynamic controlled outboxMessage: Name -> MessageType
    dynamic controlled outboxName: Name -> Name
    dynamic controlled outboxMoney: Name -> MoneyDomain
    dynamic controlled outboxTid: Name -> TidDomain
    dynamic controlled outboxIsNone: Name -> Boolean
    derived isNone: Name -> Boolean
    static authentic: Name -> Boolean //true if the card is authentic

definitions:
    domain TidDomain = {1n..2n}
    domain MoneyDomain = {0n, 5n, 10n}

    function isNone($name in Name) =
        outboxIsNone($name) or outboxMessage($name) = ACK

    function authentic($n in Name) = if($n = AA or $n = BB) then
                        true
                    else
                        false
                    endif

    macro rule r_startTo($receiver in Name) =
        //receiver cannot be involved in a previous transaction
        if(isNone($receiver)) then
            choose $na in Name, $value in MoneyDomain, $tid in TidDomain
    with
                    not(tids($tid)) //we must use a new tid
                    and authentic($na) //$na must be an authentic card
                    and $na != $receiver
                    do
            par
                inbox($na, REQ, $receiver, $value, $tid) := true
                outboxMessage($receiver) := REQ
                outboxName($receiver) := $na
                outboxMoney($receiver) := $value
                outboxTid($receiver) := $tid
                outboxIsNone($receiver) := false
                tids($tid) := true
            endpar
        endif

    macro rule r_req($receiver in Name) =
        choose $na in Name, $value in MoneyDomain, $tid in TidDomain with
                inbox($receiver, REQ, $na, $value, $tid)
                and authentic($na)
                and $na!=$receiver
                and $value <= balance($receiver)
                and (isNone($receiver))//receiver cannot be involved in
                               //a previous transaction
                do
            par
                inbox($na, VAL, $receiver, $value, $tid) := true
            outboxMessage($receiver) := VAL
            outboxName($receiver) := $na
            outboxMoney($receiver) := $value
            outboxTid($receiver) := $tid
            outboxIsNone($receiver) := false
            balance($receiver) := balance($receiver) - $value
            inbox($receiver, REQ, $na, $value, $tid) := false
        endpar

macro rule r_val($receiver in Name) =
    choose $na in Name, $value in MoneyDomain, $tid in TidDomain with
            inbox($receiver, VAL, $na, $value, $tid) and
            (outboxIsNone($receiver) = false and
            outboxMessage($receiver) = REQ and
            outboxName($receiver) = $na and
            outboxMoney($receiver) = $value and
            outboxTid($receiver) = $tid) do
        par
            inbox($na, ACK, $receiver, $value, $tid) := true
            outboxMessage($receiver) := ACK
            outboxName($receiver) := $na
            outboxMoney($receiver) := $value
            outboxTid($receiver) := $tid
            outboxIsNone($receiver) := false
            balance($receiver) := balance($receiver) + $value
            inbox($receiver, VAL, $na, $value, $tid) := false
        endpar

macro rule r_ack($receiver in Name) =
    choose $na in Name, $value in MoneyDomain, $tid in TidDomain with
            inbox($receiver, ACK, $na, $value, $tid) and
            (outboxIsNone($receiver) = false and
            outboxMessage($receiver) = VAL and
            outboxName($receiver) = $na and
            outboxMoney($receiver) = $value and
            outboxTid($receiver) = $tid) do
        par
            outboxIsNone($receiver) := true
            inbox($receiver, ACK, $na, $value, $tid) := false
        endpar

axiom over inbox: ag(inbox(AA, REQ, BB, 0n, 1n) implies
                    ef(inbox(BB, VAL, AA, 0n, 1n)))
axiom over inbox: not(ef(inbox(AA, REQ, BB, 0n, 1n) and
                                            inbox(BB, REQ, AA, 0n, 2n)))
axiom over inbox: ag((inbox(AA,REQ,BB,0n,1n) and inbox(BB,REQ,AA,0n,2n))
                              implies
          ag(not(inbox(BB,VAL,AA,0n, 1n) and inbox(AA,VAL,BB,0n,2n))))

main rule r_irule =
    choose $receiver in Name, $rule in RuleId with authentic($receiver)
do
        switch($rule)
            case STARTTORULE:
                r_startTo[$receiver]
            case REQRULE:
                r_req[$receiver]
            case VALRULE:
                r_val[$receiver]
            case ACKRULE:
                r_ack[$receiver]
        endswitch
```

```
default init s0:
    function balance($n in Name) = at({AA->5n, BB->5n},$n)
    function inbox($n in Name, $t in MessageType, $na in Name, $value in
    MoneyDomain,$tid in TidDomain) = false
    function tids($tid in TidDomain) = false
    function outboxIsNone($n in Name) = true
    function outboxMessage($n in Name) = ACK
    function outboxName($n in Name) = AA
    function outboxMoney($n in Name) = 0n
    function outboxTid($n in Name) = 1n
```

Code 6.10 is the NuSMV code obtained from the mapping of AsmetaL code 6.9.

Code 6.10: Mondex protocol with error: partial NuSMV model

```
MODULE main
    VAR
        balance_AA: {0, 10, 5};
        balance_BB: {0, 10, 5};
        inbox_AA_ACK_AA_0_1: boolean;
        .
        .
        .
        inbox_BB_VAL_BB_5_2: boolean;
        outboxIsNone_AA: boolean;
        .
        .
        .
        outboxTid_BB: 1..2;
        tids_1: boolean;
        tids_2: boolean;
        var_$na_1: {AA, BB};
        .
        .
        .
        var_$value_4: {0, 10, 5};
    DEFINE
        isNone_AA := (outboxIsNone_AA) | (outboxMessage_AA = ACK);
        isNone_BB := (outboxIsNone_BB) | (outboxMessage_BB = ACK);
        authentic_AA := TRUE;
        authentic_BB := TRUE;
    ASSIGN
        init(balance_AA) := 5;
        init(balance_BB) := 5;
        init(inbox_AA_ACK_AA_0_1) := FALSE;
        .
        .
        .
        init(inbox_BB_VAL_BB_5_2) := FALSE;
        init(outboxIsNone_AA) := TRUE;
        init(outboxIsNone_BB) := TRUE;
        init(outboxMessage_AA) := ACK;
        init(outboxMessage_BB) := ACK;
        init(outboxMoney_AA) := 0;
        init(outboxMoney_BB) := 0;
        init(outboxName_AA) := AA;
        init(outboxName_BB) := AA;
        init(outboxTid_AA) := 1;
        init(outboxTid_BB) := 1;
        init(tids_1) := FALSE;
        init(tids_2) := FALSE;

        --the following four initializations have been added by us
        --in the code obtained from the mapping. In this way
        --we can choose a particular initial state: BB executes
        --STARTTORULE and asks to AA 0 money; the tid of the
        --transaction is 1.
        init(var_$receiver_0) := BB; --added after the mapping
        init(var_$rule_0) := STARTTORULE; --added after the mapping
        init(var_$na_1) := AA; --added after the mapping
        init(var_$tid_1) := 1; --added after the mapping
        init(var_$value_1) := 0; --added after the mapping

        next(balance_AA) :=
            .
            .
            .

SPEC    AG(inbox_AA_REQ_BB_0_1 -> EF(inbox_BB_VAL_AA_0_1));
SPEC    !EF((inbox_AA_REQ_BB_0_1) & (inbox_BB_REQ_AA_0_2));
SPEC    AG((inbox_AA_REQ_BB_0_1) & (inbox_BB_REQ_AA_0_2) -> AG(!((
    inbox_BB_VAL_AA_0_1) | (inbox_AA_VAL_BB_0_2))));
```

We don't show the complete NuSMV code because it's too big (668 lines); we just show the declarations, the initializations and the properties.
In the NuSMV code 6.10 we have added the initialization of choose variables (*var_$receiver_0*, *var_$rule_0*, *var_$na_1*, *var_$tid_1*, *var_$value_1*) in order to let the model start with the execution of *r_startTo* rule in which card *BB* asks 0 money to *AA*; the tid of the transaction is 1.
Let's see the execution of NuSMV code:

```
C:\code>NuSMV -dynamic -coi mcCap3ForNuSMVoutboxSing.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG (inbox_AA_REQ_BB_0_1 -> EF inbox_BB_VAL_AA_0_1)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  balance_AA = 5
  balance_BB = 5
  inbox_AA_ACK_AA_0_1 = 0
  .
  .
  .
  inbox_BB_VAL_BB_5_2 = 0
  outboxIsNone_AA = 1
  outboxIsNone_BB = 1
  outboxMessage_AA = ACK
  outboxMessage_BB = ACK
  outboxMoney_AA = 0
  outboxMoney_BB = 0
  outboxName_AA = AA
  outboxName_BB = AA
  outboxTid_AA = 1
  outboxTid_BB = 1
  tids_1 = 0
  tids_2 = 0
  var_$na_1 = AA
  var_$na_2 = BB
  var_$na_3 = BB
  var_$na_4 = BB
  var_$receiver_0 = BB
  var_$rule_0 = STARTTORULE
  var_$tid_1 = 1
  var_$tid_2 = 1
  var_$tid_3 = 1
  var_$tid_4 = 1
  var_$value_1 = 0
  var_$value_2 = 5
  var_$value_3 = 5
  var_$value_4 = 5
  authentic_BB = 1
  authentic_AA = 1
  isNone_BB = 1
  isNone_AA = 1
-> Input: 1.2 <-
-> State: 1.2 <-
  inbox_AA_REQ_BB_0_1 = 1
  outboxIsNone_BB = 0
  outboxMessage_BB = REQ
  tids_1 = 1
  var_$na_1 = BB
  var_$receiver_0 = AA
  var_$tid_1 = 2
  var_$value_1 = 5
  isNone_BB = 0
-- specification !(EF (inbox_AA_REQ_BB_0_1 & inbox_BB_REQ_AA_0_2))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  balance_AA = 5
  balance_BB = 5
  inbox_AA_ACK_AA_0_1 = 0
  .
  .
  .
  inbox_BB_VAL_BB_5_2 = 0
  outboxIsNone_AA = 1
  outboxIsNone_BB = 1
  outboxMessage_AA = ACK
  outboxMessage_BB = ACK
  outboxMoney_AA = 0
  outboxMoney_BB = 0
  outboxName_AA = AA
  outboxName_BB = AA
  outboxTid_AA = 1
  outboxTid_BB = 1
  tids_1 = 0
  tids_2 = 0
  var_$na_1 = AA
  var_$na_2 = BB
  var_$na_3 = BB
  var_$na_4 = BB
  var_$receiver_0 = BB
  var_$rule_0 = STARTTORULE
  var_$tid_1 = 1
  var_$tid_2 = 1
  var_$tid_3 = 1
  var_$tid_4 = 1
  var_$value_1 = 0
  var_$value_2 = 5
  var_$value_3 = 5
  var_$value_4 = 5
  authentic_BB = 1
  authentic_AA = 1
  isNone_BB = 1
  isNone_AA = 1
-> Input: 2.2 <-
-> State: 2.2 <-
  inbox_AA_REQ_BB_0_1 = 1
  outboxIsNone_BB = 0
  outboxMessage_BB = REQ
  tids_1 = 1
  var_$na_1 = BB
  var_$receiver_0 = AA
  var_$tid_1 = 2
  isNone_BB = 0
-> Input: 2.3 <-
-> State: 2.3 <-
  inbox_BB_REQ_AA_0_2 = 1
  outboxIsNone_AA = 0
  outboxMessage_AA = REQ
  outboxName_AA = BB
  outboxTid_AA = 2
  tids_2 = 1
  var_$receiver_0 = BB
  var_$rule_0 = VALRULE
  var_$tid_1 = 1
  var_$value_1 = 5
  isNone_AA = 0
-- specification AG ((inbox_AA_REQ_BB_0_1 & inbox_BB_REQ_AA_0_2) ->
AG !(inbox_BB_VAL_AA_0_1 | inbox_AA_VAL_BB_0_2))  is true
```

As we have say previously, the first property is false. In state 1.2 *BB* has requested 0 money to *AA* ($inbox\_AA\_REQ\_BB\_0\_1 = 1$); from the observation of the variables we can say that in the next state *AA* will request 5 money to *BB*. From this state it's not possible that the card *AA* replies to the request contained in $inbox\_AA\_REQ\_BB\_0\_1$: both the outboxes of *AA* and *BB*, in fact, are occupied by a message and the r_req rule can be executed only if the outbox of the receiver is empty.
The second property verifies that exist a state in which both cards have done a request to the other card[4]; the third property verifies that from this state it's impossible that at least one card responds to the request of the other card.

6.3.2 First solution

In the AsmetaL model described in section 6.3.1 we haven't considered the ABORT rule[5] described in chapter 3 of [15]. We think that, even without the ABORT rule, the transfer of money between a card to another card would be always satisfied if the balances of the cards permits it: in section 6.3.1, instead, we have seen that there is a situation in which two cards enter in a deadlock state when they make correct requests of money one another.
We have seen that the reintroduction of the ABORT rule can resolve the problem. In the AsmetaL code 6.11 we show only the modification we have made to the AsmetaL code 6.9.

Code 6.11: Mondex protocol: AsmetaL model with abort rule

```
asm mcCap3ForNuSMVoutboxSingWithAbort
import ./StandardLibrary

signature:
    enum domain RuleId = {STARTTORULE | REQRULE | VALRULE | ACKRULE |
    ABORTRULE}
    .
    .
    .
    dynamic controlled exLogFrom: Prod(Name, Name, MoneyDomain, TidDomain)
    -> Boolean
```

[4] The property is negated in order to obtain an example: we see that in state 2.3 $inbox\_AA\_REQ\_BB\_0\_1 = 1$ and $inbox\_BB\_REQ\_AA\_0\_2 = 1$.

[5] We haven't considered the LOSEMSG rule too, but it's not important in our discussion.

```
    dynamic controlled exLogTo: Prod(Name, Name, MoneyDomain, TidDomain) ->
    Boolean

definitions:

    .
    .
    .

    macro rule r_abort($receiver in Name) =
        par
            choose $na in Name, $value in MoneyDomain, $tid in TidDomain
    with
                    ($na=outboxName($receiver) and
                     $value=outboxMoney($receiver) and
                     $tid=outboxTid($receiver)) do
                par
                    if(outboxMessage($receiver)=REQ) then
                        exLogTo($receiver, $na, $value, $tid) := true
                    endif
                    if(outboxMessage($receiver)=VAL) then
                        exLogFrom($receiver, $na, $value, $tid) := true
                    endif
                endpar
            outboxIsNone($receiver) := true
        endpar

    axiom over inbox: ag(inbox(AA, REQ, BB, 0n, 1n) implies
                  ef(inbox(BB, VAL, AA, 0n, 1n) or exLogTo(BB, AA, 0n, 1n)))

    main rule r_irule =
        choose $receiver in Name, $rule in RuleId with authentic($receiver)
    do
            switch($rule)
                case STARTTORULE:
                    r_startTo[$receiver]
                case REQRULE:
                    r_req[$receiver]
                case VALRULE:
                    r_val[$receiver]
                case ACKRULE:
                    r_ack[$receiver]
                case ABORTRULE:
                    r_abort[$receiver]
            endswitch

default init s0:
    .
    .
    .
    function exLogFrom($n in Name, $na in Name, $value in MoneyDomain,$tid
    in TidDomain) = false
    function exLogTo($n in Name, $na in Name, $value in MoneyDomain,$tid in
    TidDomain) = false
```

The boolean functions *exLogTo($n in Name, $na in Name, $value in MoneyDomain, $tid in TidDomain)* and *exLogFrom($n in Name, $na in Name, $value in MoneyDomain, $tid in TidDomain)* contain the logs of the aborted transitions. The arguments of these functions are:

- *$n*: the owner of the log;
- *$na*: the addressee of the message;
- *$value*: the amount of money involved in the transfer;
- *$t*: the identifier of the transaction.

In the new model, the main rule can execute also the *r_abort($receiver)* rule that interrupts the transaction of money identified by the message contained in the outbox of *$receiver*.
The *r_abort* rule memorizes the message contained in its outbox in the *exLogTo* function, if it's a REQ message, or in the *exLogFrom*, if it's a VAL message. In both cases the outbox is cleared.
We can see that this new rule can resolve the problem previously described. Let's remember the example described in section 6.3.1 in which two cards enter in a deadlock state; we also describe how they can exit from this state:

1. card *BB* executes the *r_startTo* rule: it asks 0 money to card *AA*. The outbox of *BB*, at the end of the rule, contains the same message it has sent to *AA*.

2. card *AA* executes the *r_startTo* rule: it asks 0 money to card *BB*. The outbox of *AA*, at the end of the rule, contains the same message it has sent to *BB*. At this point the two cards are in deadlock.

3. Now card *BB* (it would be the same with card *AA*) executes the *r_abort* rule: it memorizes the REQ message contained in its outbox in the *exLogTo* function and clears the outbox.

4. At this point, since the outbox of *BB* is empty, the card *BB* can execute the *r_req* rule and satisfy the request received from the card *AA*.

The property

```
ag(inbox(AA, REQ, BB, 0n, 1n) implies
ef(inbox(BB, VAL, AA, 0n, 1n) or exLogTo(BB, AA, 0n, 1n)))
```

verifies that, if the card *BB* requests 0 money to *AA*, sooner or later *BB* will receive the VAL message or will abort the transaction recording the message in the *exLogTo* function.
Let's verify the correctness of the property through the execution of the NuSMV code:

```
> NuSMV -dynamic -coi mcCap3ForNuSMVoutboxSingWithAbort.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG (inbox(AA,REQ,BB,0,1) -> EF (inbox(BB,VAL,AA,0,1) |
exLogTo(BB,AA,0,1)))  is true
```

6.3.3 Second solution

In this section we present our proposal to solve the problem described in section 6.3.1.
Code 6.12 contains the modification we have made to the original code 6.9.

Code 6.12: Mondex protocol with our modification: AsmetaL model

```
asm mcCap3ForNuSMVoutboxSingCheckOnStart
import ./StandardLibrary
import ./CTLlibrary

signature:
  .
  .
  .
  derived check: Prod(Name, Name) -> Boolean


definitions:

  function check ($receiver in Name, $na in Name) =
    if(not(exist $v in MoneyDomain, $t in TidDomain with inbox($receiver,
    REQ, $na, $v, $t))) then
      true
    else
      false
    endif
  .
  .
  .

  macro rule r_startTo($receiver in Name) =
    if(isNone($receiver)) then
      choose $na in Name, $value in MoneyDomain, $tid in TidDomain
           with not(tids($tid)) and authentic($na) and $na!= $receiver
           and check($receiver, $na) do
      par
        inbox($na, REQ, $receiver, $value, $tid) := true
        outboxMessage($receiver) := REQ
        outboxName($receiver) := $na
        outboxMoney($receiver) := $value
        outboxTid($receiver) := $tid
        outboxIsNone($receiver) := false
        tids($tid) := true
      endpar
    endif
```

```
//If a card send a REQ message, sooner or later it will receive the VAL
  message.
axiom over inbox: ag(inbox(AA, REQ, BB, 0n, 1n) implies ef(inbox(BB, VAL,
  AA, 0n, 1n)))
axiom over inbox: ag(inbox(BB, REQ, AA, 0n, 1n) implies ef(inbox(AA, VAL,
  BB, 0n, 1n)))
axiom over inbox: ag(inbox(AA, REQ, BB, 0n, 2n) implies ef(inbox(BB, VAL,
  AA, 0n, 2n)))
axiom over inbox: ag(inbox(BB, REQ, AA, 0n, 2n) implies ef(inbox(AA, VAL,
  BB, 0n, 2n)))
//The next properties, since verify messages with money greater than 0,
  must also check the balance:
//if a card send a valid REQ message (to a card that have enough money),
  sooner or later it will receive the VAL message
axiom over inbox: ag((inbox(AA, REQ, BB, 5n, 1n) and ef( balance(AA) >= 5n
  )) implies ef(inbox(BB, VAL, AA, 5n, 1n)))
axiom over inbox: ag((inbox(BB, REQ, AA, 5n, 1n) and ef( balance(BB) >= 5n
  )) implies ef(inbox(AA, VAL, BB, 5n, 1n)))
axiom over inbox: ag((inbox(AA, REQ, BB, 5n, 2n) and ef( balance(AA) >= 5n
  )) implies ef(inbox(BB, VAL, AA, 5n, 2n)))
axiom over inbox: ag((inbox(BB, REQ, AA, 5n, 2n) and ef( balance(BB) >= 5n
  )) implies ef(inbox(AA, VAL, BB, 5n, 2n)))
axiom over inbox: ag((inbox(AA, REQ, BB, 10n, 1n) and ef( balance(AA) >=
  10n)) implies ef(inbox(BB, VAL, AA, 10n, 1n)))
axiom over inbox: ag((inbox(BB, REQ, AA, 10n, 1n) and ef( balance(BB) >=
  10n)) implies ef(inbox(AA, VAL, BB, 10n, 1n)))
axiom over inbox: ag((inbox(AA, REQ, BB, 10n, 2n) and ef( balance(AA) >=
  10n)) implies ef(inbox(BB, VAL, AA, 10n, 2n)))
axiom over inbox: ag((inbox(BB, REQ, AA, 10n, 2n) and ef( balance(BB) >=
  10n)) implies ef(inbox(AA, VAL, BB, 10n, 2n)))

//A REQ message remains in the inbox of a card until the VAL message is
  put in the inbox of the other card.
axiom over inbox: ag(inbox(AA, REQ, BB, 0n, 1n) implies e(inbox(AA, REQ,
  BB, 0n, 1n), inbox(BB, VAL, AA, 0n, 1n)))
axiom over inbox: ag(inbox(BB, REQ, AA, 0n, 1n) implies e(inbox(BB, REQ,
  AA, 0n, 1n), inbox(AA, VAL, BB, 0n, 1n)))
axiom over inbox: ag(inbox(AA, REQ, BB, 0n, 2n) implies e(inbox(AA, REQ,
  BB, 0n, 2n), inbox(BB, VAL, AA, 0n, 2n)))
axiom over inbox: ag(inbox(BB, REQ, AA, 0n, 2n) implies e(inbox(BB, REQ,
  AA, 0n, 2n), inbox(AA, VAL, BB, 0n, 2n)))
//The next properties, since verify messages with money greater than 0,
  must also check the balance:
//a valid REQ message (to a card that have enough money) remains in the
  inbox of a card until the VAL message is put in the inbox of the other
  card.
axiom over inbox: ag((inbox(AA, REQ, BB, 5n, 1n) and (balance(AA)>=5n))
  implies e(inbox(AA, REQ, BB, 5n, 1n), inbox(BB, VAL, AA, 5n, 1n)))
axiom over inbox: ag((inbox(BB, REQ, AA, 5n, 1n) and (balance(BB)>=5n))
  implies e(inbox(BB, REQ, AA, 5n, 1n), inbox(AA, VAL, BB, 5n, 1n)))
axiom over inbox: ag((inbox(AA, REQ, BB, 5n, 2n) and (balance(AA)>=5n))
  implies e(inbox(AA, REQ, BB, 5n, 2n), inbox(BB, VAL, AA, 5n, 2n)))
axiom over inbox: ag((inbox(BB, REQ, AA, 5n, 2n) and (balance(BB)>=5n))
  implies e(inbox(BB, REQ, AA, 5n, 2n), inbox(AA, VAL, BB, 5n, 2n)))
axiom over inbox: ag((inbox(AA, REQ, BB, 10n, 1n) and (balance(AA)>=10n))
  implies e(inbox(AA, REQ, BB, 10n, 1n), inbox(BB, VAL, AA, 10n, 1n)))
axiom over inbox: ag((inbox(BB, REQ, AA, 10n, 1n) and (balance(BB)>=10n))
  implies e(inbox(BB, REQ, AA, 10n, 1n), inbox(AA, VAL, BB, 10n, 1n)))
axiom over inbox: ag((inbox(AA, REQ, BB, 10n, 2n) and (balance(AA)>=10n))
  implies e(inbox(AA, REQ, BB, 10n, 2n), inbox(BB, VAL, AA, 10n, 2n)))
axiom over inbox: ag((inbox(BB, REQ, AA, 10n, 2n) and (balance(BB)>=10n))
  implies e(inbox(BB, REQ, AA, 10n, 2n), inbox(AA, VAL, BB, 10n, 2n)))
```

```
//If a card send a VAL message, sooner or later it will receive the ACK
  message.
axiom over inbox: ag(inbox(AA, VAL, BB, 0n, 1n) implies ef(inbox(BB, ACK,
  AA, 0n, 1n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 0n, 1n) implies ef(inbox(AA, ACK,
  BB, 0n, 1n)))
axiom over inbox: ag(inbox(AA, VAL, BB, 0n, 2n) implies ef(inbox(BB, ACK,
  AA, 0n, 2n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 0n, 2n) implies ef(inbox(AA, ACK,
  BB, 0n, 2n)))
axiom over inbox: ag(inbox(AA, VAL, BB, 5n, 1n) implies ef(inbox(BB, ACK,
  AA, 5n, 1n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 5n, 1n) implies ef(inbox(AA, ACK,
  BB, 5n, 1n)))
axiom over inbox: ag(inbox(AA, VAL, BB, 5n, 2n) implies ef(inbox(BB, ACK,
  AA, 5n, 2n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 5n, 2n) implies ef(inbox(AA, ACK,
  BB, 5n, 2n)))
axiom over inbox: ag(inbox(AA, VAL, BB, 10n, 1n) implies ef(inbox(BB, ACK,
   AA, 10n, 1n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 10n, 1n) implies ef(inbox(AA, ACK,
   BB, 10n, 1n)))
axiom over inbox: ag(inbox(AA, VAL, BB, 10n, 2n) implies ef(inbox(BB, ACK,
   AA, 10n, 2n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 10n, 2n) implies ef(inbox(AA, ACK,
   BB, 10n, 2n)))

//A VAL message remains in the inbox of a card until the ACK message is
  put in the inbox of the other card.
axiom over inbox: ag(inbox(AA, VAL, BB, 0n, 1n) implies e(inbox(AA, VAL,
  BB, 0n, 1n), inbox(BB, ACK, AA, 0n, 1n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 0n, 1n) implies e(inbox(BB, VAL,
  AA, 0n, 1n), inbox(AA, ACK, BB, 0n, 1n)))
axiom over inbox: ag(inbox(AA, VAL, BB, 0n, 2n) implies e(inbox(AA, VAL,
  BB, 0n, 2n), inbox(BB, ACK, AA, 0n, 2n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 0n, 2n) implies e(inbox(BB, VAL,
  AA, 0n, 2n), inbox(AA, ACK, BB, 0n, 2n)))
axiom over inbox: ag(inbox(AA, VAL, BB, 5n, 1n) implies e(inbox(AA, VAL,
  BB, 5n, 1n), inbox(BB, ACK, AA, 5n, 1n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 5n, 1n) implies e(inbox(BB, VAL,
  AA, 5n, 1n), inbox(AA, ACK, BB, 5n, 1n)))
axiom over inbox: ag(inbox(AA, VAL, BB, 5n, 2n) implies e(inbox(AA, VAL,
  BB, 5n, 2n), inbox(BB, ACK, AA, 5n, 2n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 5n, 2n) implies e(inbox(BB, VAL,
  AA, 5n, 2n), inbox(AA, ACK, BB, 5n, 2n)))
axiom over inbox: ag(inbox(AA, VAL, BB, 10n, 1n) implies e(inbox(AA, VAL,
  BB, 10n, 1n), inbox(BB, ACK, AA, 10n, 1n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 10n, 1n) implies e(inbox(BB, VAL,
  AA, 10n, 1n), inbox(AA, ACK, BB, 10n, 1n)))
axiom over inbox: ag(inbox(AA, VAL, BB, 10n, 2n) implies e(inbox(AA, VAL,
  BB, 10n, 2n), inbox(BB, ACK, AA, 10n, 2n)))
axiom over inbox: ag(inbox(BB, VAL, AA, 10n, 2n) implies e(inbox(BB, VAL,
  AA, 10n, 2n), inbox(AA, ACK, BB, 10n, 2n)))


main rule r_irule =
  choose $receiver in Name, $rule in RuleId with authentic($receiver) do
    switch($rule)
      case STARTTORULE:
        r_startTo[$receiver]
      case REQRULE:
          r_req[$receiver]
        case VALRULE:
          r_val[$receiver]
        case ACKRULE:
          r_ack[$receiver]
      endswitch

default init s0:
  function balance($n in Name) = at({AA->5n, BB->5n},$n)
  function inbox($n in Name, $t in MessageType, $na in Name, $value in
    MoneyDomain,$tid in TidDomain) = false
  function tids($tid in TidDomain) = false
  function outboxIsNone($n in Name) = true
  function outboxMessage($n in Name) = ACK
  function outboxName($n in Name) = AA
  function outboxMoney($n in Name) = 0n
  function outboxTid($n in Name) = 1n
```

We have added the function *check($receiver in Name, $na in Name)* that verifies that card *$receiver* has not received a request from card *$na*.
In the *r_startTo* rule, the *check(card1, card2)* function let *card1* make a request to *card2* only if *card2* has not previously made a request to *card1*. This control permits to avoid the deadlock states described in section 6.3.1. We have declared the same property that failed in the original model[6] (section 6.3.1) and that, instead, in our model is satisfied.
Moreover we have declared other properties to check the overall execution of the protocol.
The 4 liveness properties

```
ag(inbox(AA, REQ, BB, 0n, 1n) implies ef(inbox(BB, VAL, AA, 0n, 1n)))
ag(inbox(BB, REQ, AA, 0n, 1n) implies ef(inbox(AA, VAL, BB, 0n, 1n)))
ag(inbox(AA, REQ, BB, 0n, 2n) implies ef(inbox(BB, VAL, AA, 0n, 2n)))
ag(inbox(BB, REQ, AA, 0n, 2n) implies ef(inbox(AA, VAL, BB, 0n, 2n)))
```

check that, if a card send a REQ message for 0 money, sooner or later it will receive the VAL message[7].
The 8 liveness properties

```
ag((inbox(AA, REQ, BB, 5n, 1n) and ef( balance(AA) >= 5n)) implies
ef(inbox(BB, VAL, AA, 5n, 1n)))
ag((inbox(BB, REQ, AA, 5n, 1n) and ef( balance(BB) >= 5n)) implies
ef(inbox(AA, VAL, BB, 5n, 1n)))
ag((inbox(AA, REQ, BB, 5n, 2n) and ef( balance(AA) >= 5n)) implies
ef(inbox(BB, VAL, AA, 5n, 2n)))
ag((inbox(BB, REQ, AA, 5n, 2n) and ef( balance(BB) >= 5n)) implies
ef(inbox(AA, VAL, BB, 5n, 2n)))
ag((inbox(AA, REQ, BB, 10n, 1n) and ef( balance(AA) >= 10n)) implies
ef(inbox(BB, VAL, AA, 10n, 1n)))
ag((inbox(BB, REQ, AA, 10n, 1n) and ef( balance(BB) >= 10n)) implies
```

[6] ag(inbox(AA, REQ, BB, 0n, 1n) implies ef(inbox(BB, VAL, AA, 0n, 1n)))

[7] The first property is the property that, as we have say few rows above, failed with the code described in section 6.3.1

```
ef(inbox(AA, VAL, BB, 10n, 1n)))
ag((inbox(AA, REQ, BB, 10n, 2n) and ef( balance(AA) >= 10n)) implies
ef(inbox(BB, VAL, AA, 10n, 2n)))
ag((inbox(BB, REQ, AA, 10n, 2n) and ef( balance(BB) >= 10n)) implies
ef(inbox(AA, VAL, BB, 10n, 2n)))
```

are similar to the previous four properties but, since verify messages with money greater than 0, must also check the balance: if a card send a valid REQ message (to a card that have enough money on the balance), sooner or later it will receive the VAL message.
Properties

```
ag(inbox(AA, REQ, BB, 0n, 1n) implies e(inbox(AA, REQ, BB, 0n, 1n),
inbox(BB, VAL, AA, 0n, 1n)))
ag(inbox(BB, REQ, AA, 0n, 1n) implies e(inbox(BB, REQ, AA, 0n, 1n),
inbox(AA, VAL, BB, 0n, 1n)))
ag(inbox(AA, REQ, BB, 0n, 2n) implies e(inbox(AA, REQ, BB, 0n, 2n),
inbox(BB, VAL, AA, 0n, 2n)))
ag(inbox(BB, REQ, AA, 0n, 2n) implies e(inbox(BB, REQ, AA, 0n, 2n),
inbox(AA, VAL, BB, 0n, 2n)))
ag((inbox(AA, REQ, BB, 5n, 1n) and (balance(AA)>=5n)) implies
e(inbox(AA, REQ, BB, 5n, 1n), inbox(BB, VAL, AA, 5n, 1n)))
ag((inbox(BB, REQ, AA, 5n, 1n) and (balance(BB)>=5n)) implies
e(inbox(BB, REQ, AA, 5n, 1n), inbox(AA, VAL, BB, 5n, 1n)))
ag((inbox(AA, REQ, BB, 5n, 2n) and (balance(AA)>=5n)) implies
e(inbox(AA, REQ, BB, 5n, 2n), inbox(BB, VAL, AA, 5n, 2n)))
ag((inbox(BB, REQ, AA, 5n, 2n) and (balance(BB)>=5n)) implies
e(inbox(BB, REQ, AA, 5n, 2n), inbox(AA, VAL, BB, 5n, 2n)))
ag((inbox(AA, REQ, BB, 10n, 1n) and (balance(AA)>=10n)) implies
e(inbox(AA, REQ, BB, 10n, 1n), inbox(BB, VAL, AA, 10n, 1n)))
ag((inbox(BB, REQ, AA, 10n, 1n) and (balance(BB)>=10n)) implies
e(inbox(BB, REQ, AA, 10n, 1n), inbox(AA, VAL, BB, 10n, 1n)))
ag((inbox(AA, REQ, BB, 10n, 2n) and (balance(AA)>=10n)) implies
e(inbox(AA, REQ, BB, 10n, 2n), inbox(BB, VAL, AA, 10n, 2n)))
ag((inbox(BB, REQ, AA, 10n, 2n) and (balance(BB)>=10n)) implies
e(inbox(BB, REQ, AA, 10n, 2n), inbox(AA, VAL, BB, 10n, 2n)))
```

verify that a REQ message remains in the inbox of a card until the VAL message is put in the inbox of the other card. We consider only valid messages, that is messages whose money value is available on the balance of the card to whom has been made the request.
Liveness properties

```
ag(inbox(AA, VAL, BB, 0n, 1n) implies ef(inbox(BB, ACK, AA, 0n, 1n)))
ag(inbox(BB, VAL, AA, 0n, 1n) implies ef(inbox(AA, ACK, BB, 0n, 1n)))
ag(inbox(AA, VAL, BB, 0n, 2n) implies ef(inbox(BB, ACK, AA, 0n, 2n)))
ag(inbox(BB, VAL, AA, 0n, 2n) implies ef(inbox(AA, ACK, BB, 0n, 2n)))
ag(inbox(AA, VAL, BB, 5n, 1n) implies ef(inbox(BB, ACK, AA, 5n, 1n)))
ag(inbox(BB, VAL, AA, 5n, 1n) implies ef(inbox(AA, ACK, BB, 5n, 1n)))
ag(inbox(AA, VAL, BB, 5n, 2n) implies ef(inbox(BB, ACK, AA, 5n, 2n)))
ag(inbox(BB, VAL, AA, 5n, 2n) implies ef(inbox(AA, ACK, BB, 5n, 2n)))
ag(inbox(AA, VAL, BB, 10n, 1n) implies ef(inbox(BB, ACK, AA, 10n, 1n)))
ag(inbox(BB, VAL, AA, 10n, 1n) implies ef(inbox(AA, ACK, BB, 10n, 1n)))
ag(inbox(AA, VAL, BB, 10n, 2n) implies ef(inbox(BB, ACK, AA, 10n, 2n)))
ag(inbox(BB, VAL, AA, 10n, 2n) implies ef(inbox(AA, ACK, BB, 10n, 2n)))
```

verify that, if a card send a VAL message, sooner or later it will receive the ACK message.
Properties

```
ag(inbox(AA, VAL, BB, 0n, 1n) implies e(inbox(AA, VAL, BB, 0n, 1n),
inbox(BB, ACK, AA, 0n, 1n)))
ag(inbox(BB, VAL, AA, 0n, 1n) implies e(inbox(BB, VAL, AA, 0n, 1n),
inbox(AA, ACK, BB, 0n, 1n)))
ag(inbox(AA, VAL, BB, 0n, 2n) implies e(inbox(AA, VAL, BB, 0n, 2n),
inbox(BB, ACK, AA, 0n, 2n)))
ag(inbox(BB, VAL, AA, 0n, 2n) implies e(inbox(BB, VAL, AA, 0n, 2n),
inbox(AA, ACK, BB, 0n, 2n)))
ag(inbox(AA, VAL, BB, 5n, 1n) implies e(inbox(AA, VAL, BB, 5n, 1n),
inbox(BB, ACK, AA, 5n, 1n)))
ag(inbox(BB, VAL, AA, 5n, 1n) implies e(inbox(BB, VAL, AA, 5n, 1n),
inbox(AA, ACK, BB, 5n, 1n)))
ag(inbox(AA, VAL, BB, 5n, 2n) implies e(inbox(AA, VAL, BB, 5n, 2n),
inbox(BB, ACK, AA, 5n, 2n)))
ag(inbox(BB, VAL, AA, 5n, 2n) implies e(inbox(BB, VAL, AA, 5n, 2n),
inbox(AA, ACK, BB, 5n, 2n)))
ag(inbox(AA, VAL, BB, 10n, 1n) implies e(inbox(AA, VAL, BB, 10n, 1n),
inbox(BB, ACK, AA, 10n, 1n)))
ag(inbox(BB, VAL, AA, 10n, 1n) implies e(inbox(BB, VAL, AA, 10n, 1n),
inbox(AA, ACK, BB, 10n, 1n)))
ag(inbox(AA, VAL, BB, 10n, 2n) implies e(inbox(AA, VAL, BB, 10n, 2n),
inbox(BB, ACK, AA, 10n, 2n)))
ag(inbox(BB, VAL, AA, 10n, 2n) implies e(inbox(BB, VAL, AA, 10n, 2n),
inbox(AA, ACK, BB, 10n, 2n)))
```

verify that a VAL message remains in the inbox of a card until the ACK message is put in the inbox of the other card.
Let's check the correctness of the properties through the execution of the NuSMV code:

```
> NuSMV -dynamic -coi mcCap3ForNuSMVoutboxSingCheckOnStart.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG (inbox(AA,REQ,BB,0,1) -> EF inbox(BB,VAL,AA,0,1))  is true
-- specification AG (inbox(BB,REQ,AA,0,1) -> EF inbox(AA,VAL,BB,0,1))  is true
-- specification AG (inbox(AA,REQ,BB,0,2) -> EF inbox(BB,VAL,AA,0,2))  is true
-- specification AG (inbox(BB,REQ,AA,0,2) -> EF inbox(AA,VAL,BB,0,2))  is true
-- specification AG ((inbox(AA,REQ,BB,5,1) & EF balance(AA) >= 5) ->
EF inbox(BB,VAL,AA,5,1))  is true
-- specification AG ((inbox(BB,REQ,AA,5,1) & EF balance(BB) >= 5) ->
EF inbox(AA,VAL,BB,5,1))  is true
-- specification AG ((inbox(AA,REQ,BB,5,2) & EF balance(AA) >= 5) ->
EF inbox(BB,VAL,AA,5,2))  is true
-- specification AG ((inbox(BB,REQ,AA,5,2) & EF balance(BB) >= 5) ->
EF inbox(AA,VAL,BB,5,2))  is true
-- specification AG ((inbox(AA,REQ,BB,10,1) & EF balance(AA) >= 10) ->
EF inbox(BB,VAL,AA,10,1))  is true
-- specification AG ((inbox(BB,REQ,AA,10,1) & EF balance(BB) >= 10) ->
EF inbox(AA,VAL,BB,10,1))  is true
-- specification AG ((inbox(AA,REQ,BB,10,2) & EF balance(AA) >= 10) ->
EF inbox(BB,VAL,AA,10,2))  is true
-- specification AG ((inbox(BB,REQ,AA,10,2) & EF balance(BB) >= 10) ->
EF inbox(AA,VAL,BB,10,2))  is true
-- specification AG (inbox(AA,REQ,BB,0,1) -> E [ inbox(AA,REQ,BB,0,1) U
inbox(BB,VAL,AA,0,1) ] )  is true
-- specification AG (inbox(BB,REQ,AA,0,1) -> E [ inbox(BB,REQ,AA,0,1) U
inbox(AA,VAL,BB,0,1) ] )  is true
-- specification AG (inbox(AA,REQ,BB,0,2) -> E [ inbox(AA,REQ,BB,0,2) U
inbox(BB,VAL,AA,0,2) ] )  is true
-- specification AG (inbox(BB,REQ,AA,0,2) -> E [ inbox(BB,REQ,AA,0,2) U
inbox(AA,VAL,BB,0,2) ] )  is true
-- specification AG ((inbox(AA,REQ,BB,5,1) & balance(AA) >= 5) ->
E [ inbox(AA,REQ,BB,5,1) U inbox(BB,VAL,AA,5,1) ] )  is true
-- specification AG ((inbox(BB,REQ,AA,5,1) & balance(BB) >= 5) ->
E [ inbox(BB,REQ,AA,5,1) U inbox(AA,VAL,BB,5,1) ] )  is true
-- specification AG ((inbox(AA,REQ,BB,5,2) & balance(AA) >= 5) ->
E [ inbox(AA,REQ,BB,5,2) U inbox(BB,VAL,AA,5,2) ] )  is true
-- specification AG ((inbox(BB,REQ,AA,5,2) & balance(BB) >= 5) ->
E [ inbox(BB,REQ,AA,5,2) U inbox(AA,VAL,BB,5,2) ] )  is true
-- specification AG ((inbox(AA,REQ,BB,10,1) & balance(AA) >= 10) ->
E [ inbox(AA,REQ,BB,10,1) U inbox(BB,VAL,AA,10,1) ] )  is true
-- specification AG ((inbox(BB,REQ,AA,10,1) & balance(BB) >= 10) ->
E [ inbox(BB,REQ,AA,10,1) U inbox(AA,VAL,BB,10,1) ] )  is true
-- specification AG ((inbox(AA,REQ,BB,10,2) & balance(AA) >= 10) ->
E [ inbox(AA,REQ,BB,10,2) U inbox(BB,VAL,AA,10,2) ] )  is true
-- specification AG ((inbox(BB,REQ,AA,10,2) & balance(BB) >= 10) ->
E [ inbox(BB,REQ,AA,10,2) U inbox(AA,VAL,BB,10,2) ] )  is true
-- specification AG (inbox(AA,VAL,BB,0,1) -> EF inbox(BB,ACK,AA,0,1))  is true
-- specification AG (inbox(BB,VAL,AA,0,1) -> EF inbox(AA,ACK,BB,0,1))  is true
-- specification AG (inbox(AA,VAL,BB,0,2) -> EF inbox(BB,ACK,AA,0,2))  is true
-- specification AG (inbox(BB,VAL,AA,0,2) -> EF inbox(AA,ACK,BB,0,2))  is true
-- specification AG (inbox(AA,VAL,BB,5,1) -> EF inbox(BB,ACK,AA,5,1))  is true
-- specification AG (inbox(BB,VAL,AA,5,1) -> EF inbox(AA,ACK,BB,5,1))  is true
-- specification AG (inbox(AA,VAL,BB,5,2) -> EF inbox(BB,ACK,AA,5,2))  is true
-- specification AG (inbox(BB,VAL,AA,5,2) -> EF inbox(AA,ACK,BB,5,2))  is true
-- specification AG (inbox(AA,VAL,BB,10,1) -> EF inbox(BB,ACK,AA,10,1))  is true
-- specification AG (inbox(BB,VAL,AA,10,1) -> EF inbox(AA,ACK,BB,10,1))  is true
-- specification AG (inbox(AA,VAL,BB,10,2) -> EF inbox(BB,ACK,AA,10,2))  is true
-- specification AG (inbox(BB,VAL,AA,10,2) -> EF inbox(AA,ACK,BB,10,2))  is true
-- specification AG (inbox(AA,VAL,BB,0,1) -> E [ inbox(AA,VAL,BB,0,1) U
inbox(BB,ACK,AA,0,1) ] )  is true
-- specification AG (inbox(BB,VAL,AA,0,1) -> E [ inbox(BB,VAL,AA,0,1) U
inbox(AA,ACK,BB,0,1) ] )  is true
-- specification AG (inbox(AA,VAL,BB,0,2) -> E [ inbox(AA,VAL,BB,0,2) U
inbox(BB,ACK,AA,0,2) ] )  is true
-- specification AG (inbox(BB,VAL,AA,0,2) -> E [ inbox(BB,VAL,AA,0,2) U
inbox(AA,ACK,BB,0,2) ] )  is true
-- specification AG (inbox(AA,VAL,BB,5,1) -> E [ inbox(AA,VAL,BB,5,1) U
inbox(BB,ACK,AA,5,1) ] )  is true
-- specification AG (inbox(BB,VAL,AA,5,1) -> E [ inbox(BB,VAL,AA,5,1) U
inbox(AA,ACK,BB,5,1) ] )  is true
-- specification AG (inbox(AA,VAL,BB,5,2) -> E [ inbox(AA,VAL,BB,5,2) U
inbox(BB,ACK,AA,5,2) ] )  is true
-- specification AG (inbox(BB,VAL,AA,5,2) -> E [ inbox(BB,VAL,AA,5,2) U
inbox(AA,ACK,BB,5,2) ] )  is true
-- specification AG (inbox(AA,VAL,BB,10,1) -> E [ inbox(AA,VAL,BB,10,1) U
inbox(BB,ACK,AA,10,1) ] )  is true
-- specification AG (inbox(BB,VAL,AA,10,1) -> E [ inbox(BB,VAL,AA,10,1) U
inbox(AA,ACK,BB,10,1) ] )  is true
-- specification AG (inbox(AA,VAL,BB,10,2) -> E [ inbox(AA,VAL,BB,10,2) U
inbox(BB,ACK,AA,10,2) ] )  is true
-- specification AG (inbox(BB,VAL,AA,10,2) -> E [ inbox(BB,VAL,AA,10,2) U
inbox(AA,ACK,BB,10,2) ] )  is true
```

6.4 Taxi central

In this section we describe a problem that we have already modeled directly in NuSMV; in this model we have verified some properties. We'll call this model *originalNuSMV*
Now, for the same problem, we create an AsmetaL model containing the same properties checked in *originalNuSMV*. We'll call the NuSMV model obtained from the mapping *mappedNuSMV*.
We'll see that the verification of the properties in *originalNuSMV* and the verification of the properties in *mappedNuSMV* give the same results. Obviously this cannot be considered a demonstration of the correctness of the mapping, but shows that, for a problem, there are different equivalent models.
We have seen that, 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 avoid in the direct encoding.
The comparison of the two codes, moreover, can be useful to write AsmetaL

models that produce better NuSMV models.

6.4.1 Problem

In a city, a taxi station (also called *central*) receives the clients requests; the central knows the state and the position of all the taxis and, so, it couples each client with a free taxi. The city is represented by a 8x8 grid of integer coordinates.
There are two types of clients:

- single client that needs one taxi;
- group of clients that needs more than one taxi.

In the city there are three single clients, a group that requires two taxis and a group that requires three taxis. Each client must do two travels.
Client can be in five states:

- IDLE: is walking or is standing in a point of the city;
- CALLTAXI: is calling the central to book a taxi; in this state the client chooses the destination coordinates of his travel;
- WAITING: is standing in a point of the grid, waiting the booked taxi;
- TRAVELLING: is travelling with the taxi;
- ENDTRAVELS: has done two travels and he doesn't have to travel anymore.

Taxi can be in three states:

- IDLE: is stopped in a point of the grid, waiting to be coupled to a client;
- TOCLIENT: is travelling towards the client who has booked it;
- WITHCLIENT: is travelling, with the client on board, towards the destination chosen by the client.

In the city there are three taxi.
We want to simulate the process of taxis booking, the travels of taxis towards the clients, and the travels of clients on board of the taxis towards the chosen destinations.
In order to simplify the model, we split the problem in two subproblems:

- in section 6.4.2 we model the service given to a single client; in this section we want to verify the correctness of the taxi and client movements. We want also verify that the client states and the taxi states are compatible. Finally we want to verify that the client can do both his two travels.
- in section 6.4.3 we model the coupling of taxis to clients; in this section we are just interested in the correctness of the booking system.

6.4.2 Taxi and client movements

Original NuSMV model Let's see the NuSMV model that we had previously developed (code 6.13).

Code 6.13: Taxi and client movements: original NuSMV model

```
MODULE moduletaxi(client)
    VAR
        state:{IDLE, TOCLIENT, WITHCLIENT};
        posX: 0..8; --coordinate x of taxi
        posY: 0..8; --coordinate y of taxi

    ASSIGN
        init(state) :=IDLE;
        init(posX) := {0..8};
        init(posY) := {0..8};
        next(state) :=case
            client.state=CALLTAXI & state=IDLE:TOCLIENT;
            state=TOCLIENT & client.posX=posX & client.posY=posY:WITHCLIENT;
            state=WITHCLIENT & posX = client.destX & posY = client.destY:
    IDLE;
            1:state;
        esac;
        next(posX) :=case
            state=TOCLIENT & posX > client.posX : posX - 1 ;
            state=TOCLIENT & posX < client.posX : posX + 1 ;
            state=WITHCLIENT & posX> client.destX: posX - 1 ;
            state=WITHCLIENT & posX< client.destX: posX + 1 ;
            1: posX;
        esac;
        next(posY) :=case
            state=TOCLIENT & posY > client.posY : posY - 1 ;
            state=TOCLIENT & posY < client.posY : posY + 1 ;
            state=WITHCLIENT & posY> client.destY: posY - 1 ;
            state=WITHCLIENT & posY< client.destY: posY + 1 ;
            1: posY;
        esac;

MODULE moduleclient(tax)
    VAR
        posX: 0..8;
        posY: 0..8;
        destX: 0..8;
        destY: 0..8;
        clientNumTravels: 0..2;
        state: {WAITING,TRAVELLING,IDLE,CALLTAXI,ENDTRAVELS};

    ASSIGN
        init(state) := IDLE;
        init(posX) := {0..8};
        init(posY) := {0..8};
        init(clientNumTravels) := 2;
        next(state) := case
            state=IDLE & clientNumTravels>0 :{IDLE, CALLTAXI};
            state=IDLE & clientNumTravels=0 :ENDTRAVELS;
            state=CALLTAXI:WAITING;
            state=WAITING & tax.posX=posX & tax.posY=posY:TRAVELLING;
            state=TRAVELLING & tax.posX=destX & tax.posY=destY: IDLE;
            1:state;
        esac;
        next(destX) :=case
            next(state)=CALLTAXI : {0..8};
            1: destX;
        esac;
        next(destY) :=case
            next(state)=CALLTAXI : {0..8};
            1: destY;
        esac;
        next(posX) :=case
            state=IDLE & posX>0 & posX < 8:{ posX -1, posX, posX +1 };
            state=TRAVELLING: next(tax.posX);
            1: posX;
        esac;
        next(posY) :=case
            state=IDLE & posY>0 & posY < 8:{ posY -1 , posY, posY +1 };
            state=TRAVELLING: next(tax.posY);
            1: posY;
        esac;

        next(clientNumTravels) :=case
            state=CALLTAXI & clientNumTravels>0 : clientNumTravels - 1;
            1: clientNumTravels;
        esac;

MODULE main
    VAR
        client: moduleclient(taxi);
        taxi: moduletaxi(client);

--safety property: it the taxi is travelling with the client their
    coordinates must be equal
SPEC    AG(taxi.state=WITHCLIENT ->(taxi.posX=client.posX & taxi.posY=client
    .posY));
--when the client has done his travels, he doesn't travel anymore
SPEC    AG(client.state=ENDTRAVELS -> AG(client.state=ENDTRAVELS));
--liviness property: if the client is waiting the taxi, the taxi sooner or
    later will arrive and the client will be able to travel
SPEC    AG(client.state=WAITING -> AF(client.state= TRAVELLING));
--safety property: the client is travelling only if the taxi is in
    WITHCLIENT
SPEC    AG(client.state = TRAVELLING <-> taxi.state = WITHCLIENT);
--safety property: when the taxi is going to the client, the client is
    waiting for it
SPEC    AG(client.state = WAITING <-> taxi.state = TOCLIENT);
--reachability property: it exists a state in which the client calls a taxi
    that is in its position
SPEC    EF((client.state = CALLTAXI & client.posX = taxi.posX) & client.posY
    = taxi.posY)
--deadlock absence
SPEC    AG(EX(TRUE))
```

The model is made of:

- module *moduletaxi* that represents the taxi;
- module *moduleclient* that represents the client;
- module *main* in which are created the instances of the two previous modules[8].

The states of the taxi and of the client, described by variables *state* in the two modules, are the same described in section 6.4.1.
Variables *posX* and *posY* of taxi and client are the coordinates of their positions.
The client, moreover, has variables *destX* and *destY* to record the coordinates of the destination of the travel, and the variable *clientNumTravels* to record the number of travels he have to do before entering state ENDTRAVELS.
The client, when is in $IDLE$, can stay in one place or move in each direction; when he's waiting a taxi ($CALLTAXI$ or $WAITING$), instead, he remains in the place where he has called the taxi. When a taxi and a client are travelling towards a destination, their coordinates are equal.
In the model we have verified some properties.
A safety property checks that, if the taxi is travelling with the client, their coordinates are equal:

`AG(taxi.state=WITHCLIENT -> (taxi.posX=client.posX & taxi.posY=client.posY))`

Another property checks that, when the client has done two travels, he doesn't travel anymore:

`AG(client.state=ENDTRAVELS -> AG(client.state=ENDTRAVELS))`

A liveness property checks that, if the client is waiting a taxi, the taxi sooner or later will arrive and the client will be able to travel:

`AG(client.state=WAITING ->  AF(client.state= TRAVELLING))`

A safety property checks that the client is travelling only if the taxy is in $WITHCLIENT$:

[8] We have created only the instances of a taxi and a client, because now we are interested in the modelling of the movements and not of the global booking system.

```
AG(client.state = TRAVELLING <-> taxi.state = WITHCLIENT)
```

A safety property checks that, when the taxi is going to the client, the client is waiting for it:

```
AG(client.state = WAITING <-> taxi.state = TOCLIENT)
```

A reachability property checks that exists a state in which the client calls a taxi that is in his position:

```
EF((client.state = CALLTAXI & client.posX = taxi.posX) &
client.posY = taxi.posY)
```

Finally we verify the absence of deadlock:

```
AG(EX(TRUE))
```

Let's check the correctness of the properties:

```
[user@localhost progetto_taxi]$ NuSMV taxi_sing.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG (taxi.state = WITHCLIENT ->
(taxi.posX = client.posX & taxi.posY = client.posY))  is true
-- specification AG (client.state = ENDTRAVELS ->
AG client.state = ENDTRAVELS)  is true
-- specification AG (client.state = WAITING ->
AF client.state = TRAVELLING)  is true
-- specification AG (client.state = TRAVELLING <->
taxi.state = WITHCLIENT)  is true
-- specification AG (client.state = WAITING <->
taxi.state = TOCLIENT)  is true
-- specification EF ((client.state = CALLTAXI & client.posX = taxi.posX) &
client.posY = taxi.posY)  is true
-- specification AG (EX TRUE)  is true
```

AsmetaL model In this section we describe the AsmetaL model we have written for the subproblem (code 6.14).

Code 6.14: Taxi and client movements: AsmetaL model

```
asm taxi_sing
import ./StandardLibrary
import ./CTLlibrary

signature:
    domain Coord subsetof Integer
    domain NumTravels subsetof Integer
    domain Step subsetof Integer
    enum domain TaxiState = {IDLETX | TOCLIENT | WITHCLIENT}
    enum domain ClientState = {WAITING | TRAVELLING | IDLECL | CALLTAXI |
    ENDTRAVELS}
    dynamic controlled taxiState: TaxiState
    dynamic controlled taxiPosX: Coord
    dynamic controlled taxiPosY: Coord
    dynamic controlled clientState: ClientState
    dynamic controlled clientPosX: Coord
    dynamic controlled clientPosY: Coord
    dynamic controlled clientDestX: Coord
    dynamic controlled clientDestY: Coord
    dynamic controlled clientNumTravels: NumTravels
    dynamic monitored stepX: Step
    dynamic monitored stepY: Step
    dynamic monitored decideToTravel: Boolean
    dynamic monitored destX: Coord
    dynamic monitored destY: Coord

definitions:
    domain Coord = {1..8}
    domain NumTravels = {0..2}
    domain Step ={-1..1}

    //updates coordinate $i of a taxi or of a client
    //that is travelling towards $j
    rule r_updateCoord($i in Coord, $j in Coord) =
        par
            if($i > $j) then
                $i := $i - 1
            endif
            if($i < $j) then
                $i := $i + 1
            endif
        endpar

    rule r_idleMove =
        par
            if(clientPosX > 1 and clientPosX < 8) then
                clientPosX := clientPosX + stepX
            endif
            if(clientPosY > 1 and clientPosY < 8) then
                clientPosY := clientPosY + stepY
            endif
        endpar

    rule r_idle =
        if(clientState=IDLECL) then
            if(clientNumTravels>0) then
                if(decideToTravel) then
                    par
                        clientState := CALLTAXI
                        clientDestX := destX
                        clientDestY := destY
                    endpar
                else
                    r_idleMove[]//if he doesn't call the taxi, he can walk
    around
                endif
            else
                clientState := ENDTRAVELS
        endif
    endif

rule r_receiveCall =
    if(clientState = CALLTAXI) then
        par
            taxiState := TOCLIENT //taxi goes to the client
            clientState := WAITING //client waits the taxi
        endpar
    endif

rule r_toClient =
    if(taxiState=TOCLIENT) then
        if(taxiPosX = clientPosX and taxiPosY = clientPosY) then
            par
                taxiState := WITHCLIENT //taxi picks up the client
                clientState := TRAVELLING //client is travelling with
taxi
            endpar
        else
            par
                r_updateCoord[taxiPosX, clientPosX]
                r_updateCoord[taxiPosY, clientPosY]
            endpar
        endif
    endif

rule r_withClient =
    if(taxiState=WITHCLIENT) then
        if(taxiPosX = clientDestX and taxiPosY = clientDestY) then
            par
                taxiState := IDLETX //taxi has no clients on board
                clientState := IDLECL //client is no more on taxi
                clientNumTravels := clientNumTravels - 1
            endpar
        else
            par
                r_updateCoord[taxiPosX, clientDestX]
                r_updateCoord[taxiPosY, clientDestY]
                r_updateCoord[clientPosX, clientDestX]
                r_updateCoord[clientPosY, clientDestY]
            endpar
        endif
    endif

//safety property: it the taxi is travelling with the
//client their coordinates must be equal
axiom over taxiState: ag(taxiState = WITHCLIENT implies (taxiPosX =
clientPosX and taxiPosY = clientPosY))

//when the client has done his travels, he don't
//travel anymore
axiom over clientState: ag((clientState = ENDTRAVELS) implies ag(
clientState = ENDTRAVELS))

//liviness property: if the client is waiting the taxi,
//the taxi sooner or later will arrive and the
//client will be able to travel
axiom over clientState: ag((clientState = WAITING) implies af(
clientState = TRAVELLING))

//safety property: the client is travelling only if
```

```
    //the taxi is in WITHCLIENT
    axiom over clientState: ag(clientState = TRAVELLING iff taxiState =
    WITHCLIENT)

    //safety property:when the taxi is going to the client,
    //the client is waiting for it
    axiom over clientState: ag(clientState = WAITING iff taxiState =
    TOCLIENT)

    //reachability property: it exists a state in which
    //the client calls a taxi that is in its position
    axiom over clientState: ef(clientState = CALLTAXI and clientPosX =
    taxiPosX and clientPosY = taxiPosY)

    //deadlock absence
    axiom over clientState: ag(ex(true))

    main rule r_Main =
        par
            r_idle[]
            r_receiveCall[]
            r_toClient[]
            r_withClient[]
        endpar

default init s0:
    function clientNumTravels = 2
    function clientState = IDLECL
    function taxiState = IDLETX
```

We can see that each variable of the NuSMV code 6.13 has an equivalent location in the AsmetaL code 6.14.
The state of the machine is determined by the four rules called in the main rule:

- *r_idle*: if the client is in $IDLE$, he can decide to call a taxi, entering state $CALLTAXI$, or to stay in $IDLE$;
- *r_receivecall*: if the client has called the taxi, the taxi receives the call and enters in state $TOCLIENT$; the client enters in state $WAITING$;
- *r_toClient*: if the taxi is travelling towards the client (state *TOCLIENT*), this rule modifies the coordinates of the taxi in order to reduce the distance from the client; if the coordinates of the taxi and of the client are equal, taxi enters in state $WITHCLIENT$ and client in state $TRAVELLING$;
- *r_withClient*: if the client and the taxi are travelling towards the destination of the travel, this rule modifies their coordinates; if taxi coordinates and client coordinates are equal, they enter in state $IDLE$.

In each transaction just one of the previous four rules can modify the state of the system.

If we look at the properties declared in the axiom section, we can see that they correspond to the properties declared in the original NuSMV code (code 6.13).

NuSMV model obtained from the mapping In this section we describe the NuSMV code obtained from the mapping of AsmetaL code 6.14 (code 6.15); we want to compare it with the NuSMV code developed by ourselves (code 6.13).

Code 6.15: Taxi and client movements: NuSMV model obtained from the mapping

```
MODULE main
    VAR
        clientDestX: 1..8;
        clientDestY: 1..8;
        clientNumTravels: 0..2;
        clientPosX: 1..8;
        clientPosY: 1..8;
        clientState: {CALLTAXI, ENDTRAVELS, IDLECL, TRAVELLING, WAITING};
        decideToTravel: boolean;
        destX: 1..8;
        destY: 1..8;
        stepX: -1..1;
        stepY: -1..1;
        taxiPosX: 1..8;
        taxiPosY: 1..8;
        taxiState: {IDLETX, TOCLIENT, WITHCLIENT};
    ASSIGN
        init(clientNumTravels) := 2;
        init(clientState) := IDLECL;
        init(taxiState) := IDLETX;
        next(clientDestX) :=
            case
                ((clientState = IDLECL) & (clientNumTravels > 0) & (next(
    decideToTravel))) & next(destX) in 1..8: next(destX);
                TRUE: clientDestX;
            esac;
        next(clientDestY) :=
            case
                ((clientState = IDLECL) & (clientNumTravels > 0) & (next(
    decideToTravel))) & next(destY) in 1..8: next(destY);
                TRUE: clientDestY;
            esac;
        next(clientNumTravels) :=
            case
                ((taxiState = WITHCLIENT) & ((taxiPosX = clientDestX) & (
    taxiPosY = clientDestY))) & clientNumTravels - 1 in 0..2:
    clientNumTravels - 1;
                TRUE: clientNumTravels;
            esac;
        next(clientPosX) :=
            case
                ((taxiState = WITHCLIENT) & (!((taxiPosX = clientDestX) & (
    taxiPosY = clientDestY))) & (clientPosX < clientDestX)) & clientPosX + 1
     in 1..8: clientPosX + 1;
```

```
            ((clientState = IDLECL) & (clientNumTravels > 0) & (!(next(
decideToTravel))) & ((clientPosX > 1) & (clientPosX < 8))) & clientPosX
+ next(stepX) in 1..8: clientPosX + next(stepX);
            ((taxiState = WITHCLIENT) & (!((taxiPosX = clientDestX) & (
taxiPosY = clientDestY))) & (clientPosX > clientDestX)) & clientPosX - 1
 in 1..8: clientPosX - 1;
            TRUE: clientPosX;
        esac;
    next(clientPosY) :=
        case
            ((clientState = IDLECL) & (clientNumTravels > 0) & (!(next(
decideToTravel))) & ((clientPosY > 1) & (clientPosY < 8))) & clientPosY
+ next(stepY) in 1..8: clientPosY + next(stepY);
            ((taxiState = WITHCLIENT) & (!((taxiPosX = clientDestX) & (
taxiPosY = clientDestY))) & (clientPosY < clientDestY)) & clientPosY + 1
 in 1..8: clientPosY + 1;
            ((taxiState = WITHCLIENT) & (!((taxiPosX = clientDestX) & (
taxiPosY = clientDestY))) & (clientPosY > clientDestY)) & clientPosY - 1
 in 1..8: clientPosY - 1;
            TRUE: clientPosY;
        esac;
    next(clientState) :=
        case
            (clientState = CALLTAXI): WAITING;
            ((clientState = IDLECL) & (clientNumTravels > 0) & (next(
decideToTravel))): CALLTAXI;
            ((taxiState = WITHCLIENT) & ((taxiPosX = clientDestX) & (
taxiPosY = clientDestY))): IDLECL;
            ((taxiState = TOCLIENT) & ((taxiPosX = clientPosX) & (
taxiPosY = clientPosY))): TRAVELLING;
            ((clientState = IDLECL) & (!(clientNumTravels > 0))):
ENDTRAVELS;
            TRUE: clientState;
        esac;
    next(taxiPosX) :=
        case
            (((taxiState = TOCLIENT) & (!((taxiPosX = clientPosX) & (
taxiPosY = clientPosY))) & (taxiPosX > clientPosX)) | ((taxiState =
WITHCLIENT) & (!((taxiPosX = clientDestX) & (taxiPosY = clientDestY))) &
 (taxiPosX > clientDestX))) & taxiPosX - 1 in 1..8: taxiPosX - 1;
            (((taxiState = TOCLIENT) & (!((taxiPosX = clientPosX) & (
taxiPosY = clientPosY))) & (taxiPosX < clientPosX)) | ((taxiState =
WITHCLIENT) & (!((taxiPosX = clientDestX) & (taxiPosY = clientDestY))) &
 (taxiPosX < clientDestX))) & taxiPosX + 1 in 1..8: taxiPosX + 1;
            TRUE: taxiPosX;
        esac;
    next(taxiPosY) :=
        case
            (((taxiState = TOCLIENT) & (!((taxiPosX = clientPosX) & (
taxiPosY = clientPosY))) & (taxiPosY > clientPosY)) | ((taxiState =
WITHCLIENT) & (!((taxiPosX = clientDestX) & (taxiPosY = clientDestY))) &
 (taxiPosY > clientDestY))) & taxiPosY - 1 in 1..8: taxiPosY - 1;
            (((taxiState = TOCLIENT) & (!((taxiPosX = clientPosX) & (
taxiPosY = clientPosY))) & (taxiPosY < clientPosY)) | ((taxiState =
WITHCLIENT) & (!((taxiPosX = clientDestX) & (taxiPosY = clientDestY))) &
 (taxiPosY < clientDestY))) & taxiPosY + 1 in 1..8: taxiPosY + 1;
            TRUE: taxiPosY;
        esac;
    next(taxiState) :=
        case
            (clientState = CALLTAXI): TOCLIENT;
            ((taxiState = WITHCLIENT) & ((taxiPosX = clientDestX) & (
    taxiPosY = clientDestY))): IDLETX;
                ((taxiState = TOCLIENT) & ((taxiPosX = clientPosX) & (
    taxiPosY = clientPosY))): WITHCLIENT;
                TRUE: taxiState;
            esac;
SPEC    AG(taxiState = WITHCLIENT -> (taxiPosX = clientPosX) & (taxiPosY =
    clientPosY));
SPEC    AG(clientState = ENDTRAVELS -> AG(clientState = ENDTRAVELS));
SPEC    AG(clientState = WAITING -> AF(clientState = TRAVELLING));
SPEC    AG(clientState = TRAVELLING <-> taxiState = WITHCLIENT);
SPEC    AG(clientState = WAITING <-> taxiState = TOCLIENT);
SPEC    EF(((clientState = CALLTAXI) & (clientPosX = taxiPosX)) & (
    clientPosY = taxiPosY));
SPEC    AG(EX(TRUE));
```

If we compare the original NuSMV code (code 6.13) with the mapped one (code 6.15), we can see that all the variables of the original model (state, current coordinates, destination coordinates, etc.) have an equivalent variable in the mapped code. Previously, in fact, we have observed that there is a correspondence between the variables of the original NuSMV model and the functions of the AsmetaL model; so, thanks to the transitive property, there is a correspondence between the variables of the original NuSMV model and the variables of the mapped NuSMV model.

Nevertheless there are some differences.

The original NuSMV code is divided into three modules; the mapped NuSMV code, instead, has just one module as the AsmetaL model.

The two codes are also different in the way they use nondeterminism. For example we know that the client, when is in $IDLE$, can decide to stay in $IDLE$ or to enter in $CALLTAXI$; in the original NuSMV code this behaviour is obtained with a very simple instruction:

```
next(state):=
    case
        state=IDLE & clientNumTravels>0: {IDLE, CALLTAXI};
        ...
    esac;
```

In AsmetaL, the choosing process has been modeled through the use of a monitored function $decideToTravel$; the decision to change state is made in the following way:

```
rule r_idle =
    if(clientState=IDLECL) then
        if(clientNumTravels>0) then
            if(decideToTravel) then
                par
                    clientState := CALLTAXI
```

...

So, in the mapped NuSMV code, the decision to change state is made with the monitored variable *decideToTravel*:

```
next(clientState):=
    case
        (clientState = IDLECL) & (clientNumTravels>0) &
        (decideToTravel): CALLTAXI;
        ...
    esac;
```

In this case the NuSMV code obtained from the mapping is not much more complicated than the original NuSMV code. However, in other situations, the overhead introduced by the mapping could be more significant.
Finally, we can see that the verification of the properties in the mapped NuSMV code (code 6.15) gives the same results of the verification of the same properties in the original NuSMV code (code 6.13):

```
> NuSMV -dynamic -coi examples\taxi_sing.smv
** This is NuSMV 2.4.0 (compiled on Sat Oct 4 10:17:49 UTC 2008)
** For more information on NuSMV see <http://nusmv.irst.itc.it>
** or email to <nusmv-users@irst.itc.it>.
** Please report bugs to <nusmv@irst.itc.it>.
-- specification AG (taxiState = WITHCLIENT -> (taxiPosX = clientPosX &
taxiPosY = clientPosY))  is true
-- specification AG (clientState = ENDTRAVELS -> AG clientState = ENDTRAVELS)  is true
-- specification AG (clientState = WAITING -> AF clientState = TRAVELLING)  is true
-- specification AG (clientState = TRAVELLING <-> taxiState = WITHCLIENT)  is true
-- specification AG (clientState = WAITING <-> taxiState = TOCLIENT)  is true
-- specification EF ((clientState = CALLTAXI & clientPosX = taxiPosX) &
clientPosY = taxiPosY)  is true
-- specification AG (EX TRUE)  is true
```

6.4.3 Booking system model

Original NuSMV model Let's see now the original NuSMV model (code 6.16).

Code 6.16: Booking system: original NuSMV model

```
MODULE taxi
    VAR
        taxis: 0..3 ; --num of available taxis
    ASSIGN
        init(taxis) :=3; --at the beginning all the taxis are available
```

```
MODULE client(tax,numtaxi)
    VAR
        travelLength: 0..5; -- length of travel
        state: {waiting, travelling, idle, calltaxi};
    ASSIGN
        init(travelLength) :={1..5};
        init(state) :=idle;
        next(travelLength) := case
            state=travelling & travelLength>0: travelLength - 1;
            state=calltaxi : {1..5};
            1: travelLength;
        esac;
        next(state) := case
            state=idle:{idle, calltaxi};
            state=calltaxi: waiting;
            state=waiting   & tax.taxis >= numtaxi: travelling;
            state=travelling & travelLength > 0: travelling;
            state=travelling & travelLength = 0: idle;
            1: idle;
        esac;
        next(tax.taxis) := case
            state = waiting & next(state)=travelling & tax.taxis >= numtaxi
    : tax.taxis - numtaxi;
            next(state)=idle & state=travelling & travelLength = 0 & tax.
    taxis <= 3 - numtaxi: tax.taxis + numtaxi;
            1 : tax.taxis ;
        esac;

FAIRNESS
    running


MODULE main
    VAR
        t: taxi;
        CL1: process client(t,1);
        CL2: process client(t,1);
        CL3: process client(t,1);
        GR2: process client(t,2); --group that needs 2 taxis
        GR3: process client(t,3); --group that needs 3 taxis

FAIRNESS
    running
--it exists a three minutes period in which there aren't available taxis
SPEC    EF (t.taxis = 0 & EX (t.taxis = 0 & EX t.taxis = 0));
--do not exists a path in which the taxis are always busy
SPEC    !EF (t.taxis = 0 & EG t.taxis = 0);
--liviness property: if there aren't available taxis, sooner or later at
    least one taxi will become available
SPEC    AG (t.taxis = 0 -> AF t.taxis > 0);
--it exists a state in which group GR3 asks all 3 taxis and it's request is
    accepted; in the following state all the 3 taxis are busy
SPEC    EF ((t.taxis = 3 & GR3.state = waiting) & EX (t.taxis = 0 & GR3.
    state = travelling))
--liviness property: if a client or a group is waiting a taxi, sooner or
    later the taxi will arrive
SPEC    AG (CL1.state = waiting -> EF CL1.state = travelling)
SPEC    AG (CL2.state = waiting -> EF CL2.state = travelling)
SPEC    AG (CL3.state = waiting -> EF CL3.state = travelling)
SPEC    AG (GR2.state = waiting -> EF GR2.state = travelling)
SPEC    AG (GR3.state = waiting -> EF GR3.state = travelling)
--reachability property: it exists a state in which all the single clients
```

```
    and group GR2 have called a taxi. The property has been negated in order
     to obtain the print of a particular example
SPEC    !EF (((CL1.state = waiting & CL2.state = waiting) & CL3.state =
    waiting) & GR2.state = waiting)
--deadlock absence
SPEC    AG(EX(TRUE))
```

In this subproblem we are no more interested in the movements of the taxi and of the client; now we want to model the booking system: we want to check that all the requests are correctly satisfied.
Clients are different instances of module *client*; the client states are the same described in section 6.4.1[9]. Variable *numtaxi* represents the number of taxis that the client needs. Variable *travelLength* models the length of the travel that the client must do; in this model, in fact, we are not interested in the position of the client, but just in the time he holds the taxi.
Since we are not interested in the taxi movements, but just that the booking system is correct, module *taxi* contains just the variable *taxis*, that is the number of available taxis.
In the *main* module are created an instance of module *taxi*, three single clients, a group that needs two taxis and a group that needs three taxis. Since the instances of *client* have been created as asynchronous processes (they have been created through the keyword *process*), we have added to the model the declaration *FAIRNESS running* so that each instance of client is chosen "infinitely often".
In the model we have defined some properties.
A reachability property checks that exists a three minutes period in which there are no available taxis:

```
EF (t.taxis = 0 & EX (t.taxis = 0 & EX t.taxis = 0))
```

A property checks that not exists a path in which the taxis are always occupied:

```
!EF (t.taxis = 0 & EG t.taxis = 0);
```

A liveness property checks that, if there aren't available taxis, sooner or later at least one taxi will become available:

```
AG (t.taxis = 0 -> AF t.taxis > 0);
```

Another property checks that exists a state in which the group GR3 asks all the three taxis and it's request is satisfied; in the following state there aren't available taxis:

[9] There is no more the state *ENDTRAVELS* because, in this model, the number of travels is not considered.

```
EF ((t.taxis = 3 & GR3.state = waiting) & EX (t.taxis = 0 &
GR3.state = travelling))
```

Five liveness properties check that, if a client or a group is waiting a taxi, sooner or later the taxi will arrive:

```
AG (CL1.state = waiting -> EF CL1.state = travelling)
AG (CL2.state = waiting -> EF CL2.state = travelling)
AG (CL3.state = waiting -> EF CL3.state = travelling)
AG (GR2.state = waiting -> EF GR2.state = travelling)
AG (GR3.state = waiting -> EF GR3.state = travelling)
```

A reachability property checks that exist a state in which all the single clients and the group GR2 have called a taxi; the property is negated in order to obtain the print of an example:

```
!EF (((CL1.state = waiting & CL2.state = waiting) &
CL3.state = waiting) & GR2.state = waiting)
```

Finally we verify the absence of deadlock:

```
AG(EX(TRUE))
```

Let's check the correctness of the properties through the execution of NuSMV code:

```
[user@localhost progetto_taxi]$ NuSMV taxi.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
-- specification EF (t.taxis = 0 & EX (t.taxis = 0 & EX t.taxis = 0))  is true
-- specification !(EF (t.taxis = 0 & EG t.taxis = 0))  is true
-- specification AG (t.taxis = 0 -> AF t.taxis > 0)  is true
-- specification EF ((t.taxis = 3 & GR3.state = waiting) & EX (t.taxis = 0 &
GR3.state = travelling))  is true
-- specification AG (CL1.state = waiting -> EF CL1.state = travelling)  is true
-- specification AG (CL2.state = waiting -> EF CL2.state = travelling)  is true
-- specification AG (CL3.state = waiting -> EF CL3.state = travelling)  is true
-- specification AG (GR2.state = waiting -> EF GR2.state = travelling)  is true
-- specification AG (GR3.state = waiting -> EF GR3.state = travelling)  is true
-- specification !(EF (((CL1.state = waiting & CL2.state = waiting) &
CL3.state = waiting) & GR2.state = waiting))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  t.taxis = 3
  CL1.travelLength = 1
  CL1.state = idle
  CL2.travelLength = 1
  CL2.state = idle
  CL3.travelLength = 1
  CL3.state = idle
  GR2.travelLength = 1
  GR2.state = idle
  GR3.travelLength = 1
  GR3.state = idle
-> Input: 1.2 <-
  _process_selector_ = CL1
  running = 0
  GR3.running = 0
  GR2.running = 0
  CL3.running = 0
  CL2.running = 0
  CL1.running = 1
-> State: 1.2 <-
  CL1.state = calltaxi
-> Input: 1.3 <-
  _process_selector_ = CL1
  running = 0
  GR3.running = 0
  GR2.running = 0
  CL3.running = 0
  CL2.running = 0
  CL1.running = 1
-> State: 1.3 <-
  CL1.state = waiting
-> Input: 1.4 <-
  _process_selector_ = CL2
  running = 0
  GR3.running = 0
  GR2.running = 0
  CL3.running = 0
  CL2.running = 1
  CL1.running = 0
-> State: 1.4 <-
  CL2.state = calltaxi
-> Input: 1.5 <-
  _process_selector_ = CL2
  running = 0
  GR3.running = 0
  GR2.running = 0
  CL3.running = 0
  CL2.running = 1
  CL1.running = 0
-> State: 1.5 <-
```

```
  CL2.state = waiting
-> Input: 1.6 <-
  _process_selector_ = CL3
  running = 0
  GR3.running = 0
  GR2.running = 0
  CL3.running = 1
  CL2.running = 0
  CL1.running = 0
-> State: 1.6 <-
  CL3.state = calltaxi
-> Input: 1.7 <-
  _process_selector_ = CL3
  running = 0
  GR3.running = 0
  GR2.running = 0
  CL3.running = 1
  CL2.running = 0
  CL1.running = 0
-> State: 1.7 <-
  CL3.state = waiting
-> Input: 1.8 <-
  _process_selector_ = GR2
  running = 0
  GR3.running = 0
  GR2.running = 1
  CL3.running = 0
  CL2.running = 0
  CL1.running = 0
-> State: 1.8 <-
  GR2.state = calltaxi
-> Input: 1.9 <-
  _process_selector_ = GR2
  running = 0
  GR3.running = 0
  GR2.running = 1
  CL3.running = 0
  CL2.running = 0
  CL1.running = 0
-> State: 1.9 <-
  GR2.state = waiting
-- specification AG (EX TRUE)  is true
```

AsmetaL model In this section we analyze the AsmetaL model that we have developed for the subproblem (code 6.17).

Code 6.17: Booking system: AsmetaL model

```
asm taxi
import ./StandardLibrary
import ./CTLlibrary

signature:
    enum domain Client = {CL1|CL2|CL3|GR2|GR3}
    enum domain State = {WAITING | TRAVELLING | IDLE | CALLTAXI}
    domain Taxi subsetof Integer
    domain Lengths subsetof Integer
    static neededTaxi: Client -> Taxi
    derived numtaxis: Taxi
    dynamic controlled state: Client -> State
    dynamic controlled travelLength: Client -> Lengths
    dynamic monitored calltaxi: Client -> Boolean
    dynamic controlled keepTaxi: Client -> Taxi
    dynamic monitored chooseLength: Lengths

definitions:
    domain Taxi = {0..3}
    domain Lengths = {1..3}
    function numtaxis = 3- keepTaxi(CL1) - keepTaxi(CL2) - keepTaxi(CL3) -
    keepTaxi(GR2) - keepTaxi(GR3)

    function neededTaxi($c in Client) =
        if($c=CL1 or $c=CL2 or $c=CL3) then
             1
        else
            if($c=GR2) then
                2
            else
                3
            endif
        endif

    rule r_waiting =
        choose $c in Client with ((state($c) = WAITING) and numtaxis>=
    neededTaxi($c)) do
            par
                state($c) := TRAVELLING
                keepTaxi($c) := neededTaxi($c)
                travelLength($c) := chooseLength
            endpar

    rule r_travel =
        forall $c in Client with state($c)=TRAVELLING do
            if(travelLength($c) = 1) then
                par
                    state($c) :=  IDLE
                    keepTaxi($c) := 0
                endpar
            else
                travelLength($c) := travelLength($c) - 1
            endif

    rule r_idle =
        forall $c in Client with state($c)=IDLE do
            if(calltaxi($c)) then
                state($c) := CALLTAXI
            endif

    rule r_callTaxi =
        forall $c in Client with state($c)=CALLTAXI do
```

```
            state($c) := WAITING

    //it exists a three minutes period in which
    //there aren't available taxis
    axiom over numtaxis: ef(numtaxis = 0 and ex(numtaxis = 0 and ex(numtaxis
     = 0)))

    //do not exists a path in which the taxis are
    //always busy
    axiom over numtaxis: not(ef(numtaxis = 0 and eg(numtaxis = 0)))

    //liviness property: if there aren't available taxis,
    //sooner or later at least one taxi will
    //become available
    axiom over numtaxis: ag(numtaxis = 0 implies af(numtaxis > 0))

    //it exists a state in which group GR3 asks all 3
    //taxis and it's request is accepted; in the
    //following state all the 3 taxis are busy
    axiom over numtaxis: ef((numtaxis = 3 and state(GR3) = WAITING) and ex(
    numtaxis = 0 and state(GR3) = TRAVELLING))

    //liviness property: if a client or a group is
    //waiting a taxi, sooner or later the taxi
    //will arrive
    axiom over state: ag(state(CL1) = WAITING implies ef(state(CL1) =
    TRAVELLING))
    axiom over state: ag(state(CL2) = WAITING implies ef(state(CL2) =
    TRAVELLING))
    axiom over state: ag(state(CL3) = WAITING implies ef(state(CL3) =
    TRAVELLING))
    axiom over state: ag(state(GR2) = WAITING implies ef(state(GR2) =
    TRAVELLING))
    axiom over state: ag(state(GR3) = WAITING implies ef(state(GR3) =
    TRAVELLING))

    //reachability property: it exists a state in which
    //all the single clients and group GR2 have called
    //a taxi. The property has been negated in order to
    //obtain the print of a particular example.
    axiom over state: not(ef(state(CL1) = WAITING and state(CL2) = WAITING
    and state(CL3) = WAITING and state(GR2) = WAITING))

    //deadlock absence
    axiom over state: ag(ex(true))

    main rule r_Main =
        par
            r_idle[]
            r_waiting[]
            r_travel[]
            r_callTaxi[]
        endpar

default init s0:
    function state($c in Client) = IDLE
    function keepTaxi($c in Client) = 0
```

As in the previous subproblem, each variable of the original NuSMV code (code 6.16) can be associated with an equivalent location in the AsmetaL

model. The AsmetaL model, moreover, contains a monitored boolean function *calltaxi*(\$*c in Client*) that signals if the client \$*c* has called a taxi.
The state of the machine is determined by the four rules called in the main rule:

- *r_idle*: if a client \$*c* is in *IDLE*, he can decide to call a taxi (location *calltaxi*(\$*c*)), entering state *CALLTAXI*, or to stay in *IDLE*;
- *r_callTaxi*: if a client has called a taxi, he enters state *WAITING* in order to wait for it;
- *r_waiting*: if a client \$*c* is in *WAITING* and the number of available taxis is greater or equal to *neededTaxi*(\$*c*), he enters state *TRAVELLING* and occupies *neededTaxi*(\$*c*) taxis; the client decides also the length of the travel through the location *travelLength*(\$*c*);
- *r_travel*: if a client \$*c* is in *TRAVELLING* and he hasn't reached his destination, he decrements of a unity the location *travelLength*(\$*c*); if he has reached the destination, instead, he enters state *IDLE* and sets *neededTaxi*(\$*c*) taxis free.

In each transaction just one of the four rules can modify the state of the system.
If we look at the properties declared in the axiom section, we can see that they correspond to the properties declared in the original NuSMV code (code 6.16).

NuSMV model obtained from the mapping In this section we describe the NuSMV code obtained from the mapping of AsmetaL code 6.17 (code 6.18); we want to compare it with the NuSMV code developed by ourselves (code 6.16).

Code 6.18: Booking system: NuSMV model obtained from the mapping

```
MODULE main
    VAR
        calltaxi_CL1: boolean;
        calltaxi_CL2: boolean;
        calltaxi_CL3: boolean;
        calltaxi_GR2: boolean;
        calltaxi_GR3: boolean;
        chooseLength: 1..3;
        keepTaxi_CL1: 0..3;
        keepTaxi_CL2: 0..3;
        keepTaxi_CL3: 0..3;
        keepTaxi_GR2: 0..3;
        keepTaxi_GR3: 0..3;
    state_CL1: {CALLTAXI, IDLE, TRAVELLING, WAITING};
    state_CL2: {CALLTAXI, IDLE, TRAVELLING, WAITING};
    state_CL3: {CALLTAXI, IDLE, TRAVELLING, WAITING};
    state_GR2: {CALLTAXI, IDLE, TRAVELLING, WAITING};
    state_GR3: {CALLTAXI, IDLE, TRAVELLING, WAITING};
    travelLength_CL1: 1..3;
    travelLength_CL2: 1..3;
    travelLength_CL3: 1..3;
    travelLength_GR2: 1..3;
    travelLength_GR3: 1..3;
    var_$c_0: {CL1, CL2, CL3, GR2, GR3};
DEFINE
    neededTaxi_CL1 :=
        case
            (!(((CL1 = CL1) | (CL1 = CL2)) | (CL1 = CL3))) & (!(CL1 =
GR2)) & 3 in 0..3: 3;
            (!(((CL1 = CL1) | (CL1 = CL2)) | (CL1 = CL3))) & (CL1 = GR2)
 & 2 in 0..3: 2;
            ((CL1 = CL1) | (CL1 = CL2)) | (CL1 = CL3) & 1 in 0..3: 1;
        esac;
    neededTaxi_CL2 :=
        case
            (!(((CL2 = CL1) | (CL2 = CL2)) | (CL2 = CL3))) & (!(CL2 =
GR2)) & 3 in 0..3: 3;
            (!(((CL2 = CL1) | (CL2 = CL2)) | (CL2 = CL3))) & (CL2 = GR2)
 & 2 in 0..3: 2;
            ((CL2 = CL1) | (CL2 = CL2)) | (CL2 = CL3) & 1 in 0..3: 1;
        esac;
    neededTaxi_CL3 :=
        case
            (!(((CL3 = CL1) | (CL3 = CL2)) | (CL3 = CL3))) & (!(CL3 =
GR2)) & 3 in 0..3: 3;
            (!(((CL3 = CL1) | (CL3 = CL2)) | (CL3 = CL3))) & (CL3 = GR2)
 & 2 in 0..3: 2;
            ((CL3 = CL1) | (CL3 = CL2)) | (CL3 = CL3) & 1 in 0..3: 1;
        esac;
    neededTaxi_GR2 :=
        case
            (!(((GR2 = CL1) | (GR2 = CL2)) | (GR2 = CL3))) & (!(GR2 =
GR2)) & 3 in 0..3: 3;
            (!(((GR2 = CL1) | (GR2 = CL2)) | (GR2 = CL3))) & (GR2 = GR2)
 & 2 in 0..3: 2;
            ((GR2 = CL1) | (GR2 = CL2)) | (GR2 = CL3) & 1 in 0..3: 1;
        esac;
    neededTaxi_GR3 :=
        case
            (!(((GR3 = CL1) | (GR3 = CL2)) | (GR3 = CL3))) & (!(GR3 =
GR2)) & 3 in 0..3: 3;
            (!(((GR3 = CL1) | (GR3 = CL2)) | (GR3 = CL3))) & (GR3 = GR2)
 & 2 in 0..3: 2;
            ((GR3 = CL1) | (GR3 = CL2)) | (GR3 = CL3) & 1 in 0..3: 1;
        esac;
    numtaxis := 3 - keepTaxi_CL1 - keepTaxi_CL2 - keepTaxi_CL3 -
keepTaxi_GR2 - keepTaxi_GR3;
ASSIGN
    init(keepTaxi_CL1) := 0;
    init(keepTaxi_CL2) := 0;
    init(keepTaxi_CL3) := 0;
    init(keepTaxi_GR2) := 0;
    init(keepTaxi_GR3) := 0;
    init(state_CL1) := IDLE;
    init(state_CL2) := IDLE;
    init(state_CL3) := IDLE;
    init(state_GR2) := IDLE;
    init(state_GR3) := IDLE;
    next(keepTaxi_CL1) :=
        case
            ((state_CL1 = TRAVELLING) & (travelLength_CL1 = 1)) & 0 in
0..3: 0;
            ((var_$c_0 = CL1) & ((state_CL1 = WAITING) & (numtaxis >=
neededTaxi_CL1))) & neededTaxi_CL1 in 0..3: neededTaxi_CL1;
            TRUE: keepTaxi_CL1;
        esac;
    next(keepTaxi_CL2) :=
        case
            ((state_CL2 = TRAVELLING) & (travelLength_CL2 = 1)) & 0 in
0..3: 0;
            ((var_$c_0 = CL2) & ((state_CL2 = WAITING) & (numtaxis >=
neededTaxi_CL2))) & neededTaxi_CL2 in 0..3: neededTaxi_CL2;
            TRUE: keepTaxi_CL2;
        esac;
    next(keepTaxi_CL3) :=
        case
            ((state_CL3 = TRAVELLING) & (travelLength_CL3 = 1)) & 0 in
0..3: 0;
            ((var_$c_0 = CL3) & ((state_CL3 = WAITING) & (numtaxis >=
neededTaxi_CL3))) & neededTaxi_CL3 in 0..3: neededTaxi_CL3;
            TRUE: keepTaxi_CL3;
        esac;
    next(keepTaxi_GR2) :=
        case
            ((state_GR2 = TRAVELLING) & (travelLength_GR2 = 1)) & 0 in
0..3: 0;
            ((var_$c_0 = GR2) & ((state_GR2 = WAITING) & (numtaxis >=
neededTaxi_GR2))) & neededTaxi_GR2 in 0..3: neededTaxi_GR2;
            TRUE: keepTaxi_GR2;
        esac;
    next(keepTaxi_GR3) :=
        case
            ((state_GR3 = TRAVELLING) & (travelLength_GR3 = 1)) & 0 in
0..3: 0;
            ((var_$c_0 = GR3) & ((state_GR3 = WAITING) & (numtaxis >=
neededTaxi_GR3))) & neededTaxi_GR3 in 0..3: neededTaxi_GR3;
            TRUE: keepTaxi_GR3;
        esac;
    next(state_CL1) :=
        case
            ((state_CL1 = TRAVELLING) & (travelLength_CL1 = 1)): IDLE;
            ((var_$c_0 = CL1) & ((state_CL1 = WAITING) & (numtaxis >=
neededTaxi_CL1))): TRAVELLING;
            (state_CL1 = CALLTAXI): WAITING;
            ((state_CL1 = IDLE) & (next(calltaxi_CL1))): CALLTAXI;
            TRUE: state_CL1;
        esac;
    next(state_CL2) :=
        case
            ((state_CL2 = IDLE) & (next(calltaxi_CL2))): CALLTAXI;
            ((state_CL2 = TRAVELLING) & (travelLength_CL2 = 1)): IDLE;
            ((var_$c_0 = CL2) & ((state_CL2 = WAITING) & (numtaxis >=
neededTaxi_CL2))): TRAVELLING;
            (state_CL2 = CALLTAXI): WAITING;
            TRUE: state_CL2;
        esac;
    next(state_CL3) :=
        case
            ((state_CL3 = TRAVELLING) & (travelLength_CL3 = 1)): IDLE;
            ((var_$c_0 = CL3) & ((state_CL3 = WAITING) & (numtaxis >=
neededTaxi_CL3))): TRAVELLING;
            (state_CL3 = CALLTAXI): WAITING;
            ((state_CL3 = IDLE) & (next(calltaxi_CL3))): CALLTAXI;
            TRUE: state_CL3;
        esac;
    next(state_GR2) :=
        case
            ((state_GR2 = TRAVELLING) & (travelLength_GR2 = 1)): IDLE;
            ((state_GR2 = IDLE) & (next(calltaxi_GR2))): CALLTAXI;
            ((var_$c_0 = GR2) & ((state_GR2 = WAITING) & (numtaxis >=
neededTaxi_GR2))): TRAVELLING;
            (state_GR2 = CALLTAXI): WAITING;
            TRUE: state_GR2;
        esac;
    next(state_GR3) :=
        case
            ((state_GR3 = TRAVELLING) & (travelLength_GR3 = 1)): IDLE;
            (state_GR3 = CALLTAXI): WAITING;
            ((var_$c_0 = GR3) & ((state_GR3 = WAITING) & (numtaxis >=
neededTaxi_GR3))): TRAVELLING;
            ((state_GR3 = IDLE) & (next(calltaxi_GR3))): CALLTAXI;
            TRUE: state_GR3;
        esac;
    next(travelLength_CL1) :=
        case
            ((var_$c_0 = CL1) & ((state_CL1 = WAITING) & (numtaxis >=
neededTaxi_CL1))) & next(chooseLength) in 1..3: next(chooseLength);
            ((state_CL1 = TRAVELLING) & (!(travelLength_CL1 = 1))) &
travelLength_CL1 - 1 in 1..3: travelLength_CL1 - 1;
            TRUE: travelLength_CL1;
        esac;
    next(travelLength_CL2) :=
        case
            ((state_CL2 = TRAVELLING) & (!(travelLength_CL2 = 1))) &
travelLength_CL2 - 1 in 1..3: travelLength_CL2 - 1;
            ((var_$c_0 = CL2) & ((state_CL2 = WAITING) & (numtaxis >=
neededTaxi_CL2))) & next(chooseLength) in 1..3: next(chooseLength);
            TRUE: travelLength_CL2;
        esac;
    next(travelLength_CL3) :=
        case
            ((var_$c_0 = CL3) & ((state_CL3 = WAITING) & (numtaxis >=
neededTaxi_CL3))) & next(chooseLength) in 1..3: next(chooseLength);
            ((state_CL3 = TRAVELLING) & (!(travelLength_CL3 = 1))) &
travelLength_CL3 - 1 in 1..3: travelLength_CL3 - 1;
            TRUE: travelLength_CL3;
        esac;
    next(travelLength_GR2) :=
        case
            ((var_$c_0 = GR2) & ((state_GR2 = WAITING) & (numtaxis >=
neededTaxi_GR2))) & next(chooseLength) in 1..3: next(chooseLength);
            ((state_GR2 = TRAVELLING) & (!(travelLength_GR2 = 1))) &
travelLength_GR2 - 1 in 1..3: travelLength_GR2 - 1;
            TRUE: travelLength_GR2;
        esac;
    next(travelLength_GR3) :=
        case
            ((var_$c_0 = GR3) & ((state_GR3 = WAITING) & (numtaxis >=
neededTaxi_GR3))) & next(chooseLength) in 1..3: next(chooseLength);
```

```
                ((state_GR3 = TRAVELLING) & (!(travelLength_GR3 = 1))) &
    travelLength_GR3 - 1 in 1..3: travelLength_GR3 - 1;
                TRUE: travelLength_GR3;
            esac;
    INVAR   ((var_$c_0 = CL1) & ((state_CL1 = WAITING) & (numtaxis >=
    neededTaxi_CL1))) | ((var_$c_0 = CL2) & ((state_CL2 = WAITING) & (
    numtaxis >= neededTaxi_CL2))) | ((var_$c_0 = CL3) & ((state_CL3 =
    WAITING) & (numtaxis >= neededTaxi_CL3))) | ((var_$c_0 = GR2) & ((
    state_GR2 = WAITING) & (numtaxis >= neededTaxi_GR2))) | ((var_$c_0 = GR3
    ) & ((state_GR3 = WAITING) & (numtaxis >= neededTaxi_GR3))) | ((!((
    state_CL1 = WAITING) & (numtaxis >= neededTaxi_CL1))) & (!((state_CL2 =
    WAITING) & (numtaxis >= neededTaxi_CL2))) & (!((state_CL3 = WAITING) & (
    numtaxis >= neededTaxi_CL3))) & (!((state_GR2 = WAITING) & (numtaxis >=
    neededTaxi_GR2))) & (!((state_GR3 = WAITING) & (numtaxis >=
    neededTaxi_GR3))));
SPEC    EF((numtaxis = 0) & (EX((numtaxis = 0) & (EX(numtaxis = 0)))));
SPEC    !(EF((numtaxis = 0) & (EG(numtaxis = 0))));
SPEC    AG(numtaxis = 0 -> AF(numtaxis > 0));
SPEC    EF(((numtaxis = 3) & (state_GR3 = WAITING)) & (EX((numtaxis = 0) & (
    state_GR3 = TRAVELLING))));
SPEC    AG(state_CL1 = WAITING -> EF(state_CL1 = TRAVELLING));
SPEC    AG(state_CL2 = WAITING -> EF(state_CL2 = TRAVELLING));
SPEC    AG(state_CL3 = WAITING -> EF(state_CL3 = TRAVELLING));
SPEC    AG(state_GR2 = WAITING -> EF(state_GR2 = TRAVELLING));
SPEC    AG(state_GR3 = WAITING -> EF(state_GR3 = TRAVELLING));
SPEC    !(EF((((state_CL1 = WAITING) & (state_CL2 = WAITING)) & (state_CL3 =
     WAITING)) & (state_GR2 = WAITING)));
SPEC    AG(EX(TRUE));
```

We can see that the NuSMV code obtained from the mapping is much more complex that the original NuSMV code; in AsmetaL code, in fact, we have used a lot of forall and choose rules, whose mappings are particularly complex.

Finally we can observe that code 6.18 and code 6.16 verify the same properties. The counterexample of the penultimate property, obviously, is not the same; it's important that both codes give a counterexample and not that the counterexamples are the same.

```
> NuSMV -dynamic -coi examples\taxi.smv
** This is NuSMV 2.4.0 (compiled on Sat Oct 4 10:17:49 UTC 2008)
** For more information on NuSMV see <http://nusmv.irst.itc.it>
** or email to <nusmv-users@irst.itc.it>.
** Please report bugs to <nusmv@irst.itc.it>.
-- specification EF (numtaxis = 0 & EX (numtaxis = 0 & EX numtaxis = 0))  is true
-- specification !(EF (numtaxis = 0 & EG numtaxis = 0))  is true
-- specification AG (numtaxis = 0 -> AF numtaxis > 0)  is true
-- specification EF ((numtaxis = 3 & state(GR3) = WAITING) & EX (numtaxis = 0 &
state(GR3) = TRAVELLING))  is true
-- specification AG (state(CL1) = WAITING -> EF state(CL1) = TRAVELLING)  is true
-- specification AG (state(CL2) = WAITING -> EF state(CL2) = TRAVELLING)  is true
-- specification AG (state(CL3) = WAITING -> EF state(CL3) = TRAVELLING)  is true
-- specification AG (state(GR2) = WAITING -> EF state(GR2) = TRAVELLING)  is true
-- specification AG (state(GR3) = WAITING -> EF state(GR3) = TRAVELLING)  is true
-- specification !(EF (((state(CL1) = WAITING & state(CL2) = WAITING) &
state(CL3) = WAITING) & state(GR2) = WAITING))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  calltaxi(CL1) = 0
  calltaxi(CL2) = 0
  calltaxi(CL3) = 0
  calltaxi(GR2) = 0
  calltaxi(GR3) = 0
  chooseLength = 1
  keepTaxi(CL1) = 0
  keepTaxi(CL2) = 0
  keepTaxi(CL3) = 0
  keepTaxi(GR2) = 0
  keepTaxi(GR3) = 0
  state(CL1) = IDLE
  state(CL2) = IDLE
  state(CL3) = IDLE
  state(GR2) = IDLE
  state(GR3) = IDLE
  travelLength(CL1) = 1
  travelLength(CL2) = 1
  travelLength(CL3) = 1
  travelLength(GR2) = 1
  travelLength(GR3) = 1
  var_$c_0 = GR3
  numtaxis = 3
  neededTaxi(GR3) = 3
  neededTaxi(GR2) = 2
  neededTaxi(CL3) = 1
  neededTaxi(CL2) = 1
  neededTaxi(CL1) = 1
-> Input: 1.2 <-
-> State: 1.2 <-
  calltaxi(CL1) = 1
  calltaxi(CL2) = 1
  calltaxi(CL3) = 1
  calltaxi(GR2) = 1
  state(CL1) = CALLTAXI
  state(CL2) = CALLTAXI
  state(CL3) = CALLTAXI
  state(GR2) = CALLTAXI
-> Input: 1.3 <-
-> State: 1.3 <-
  calltaxi(CL1) = 0
  calltaxi(CL2) = 0
  calltaxi(CL3) = 0
  calltaxi(GR2) = 0
  calltaxi(GR3) = 1
  state(CL1) = WAITING
  state(CL2) = WAITING
  state(CL3) = WAITING
  state(GR2) = WAITING
  state(GR3) = CALLTAXI
  var_$c_0 = GR2
-- specification AG (EX TRUE)  is true
```

Bibliography

[1] The ASMETA website. `http://asmeta.sourceforge.net/`, 2006.

[2] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Ten reasons to metamodel ASMs. In *Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis, LNCS Festschrift*. Springer, 2007.

[3] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Metamodelling a Formal Method: Applying MDE to Abstract State Machines. Technical Report 97, DTI Dept., University of Milan, 2006.

[4] P. Scandurra, A. Gargantini, C. Genovese, T. Genovese, and E. Riccobene. A concrete syntax derived from the abstract state machine metamodel. In *12th International Workshop on Abstract State Machines (ASM'05)*, 8-11 March 2005, Paris, France, 2005.

[5] 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.

[6] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A metamodel-based simulator for ASMs. In Andreas Prinz, editor, *Proceedings of the 14th International ASM Workshop*, 2007.

[7] 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.

[8] ATGT: ASM Tests Generation Tool. `http://cs.unibg.it/gargantini/software/atgt`, 2008.

[9] E. Börger. R. Stärk. *Abstract State Machine. A metod for high-level system design and analysis.* Springer, 2003.

[10] Asmm concrete syntax (asmetal) v.2.0.0 - a quick guide. `http://fmlab.dti.unimi.it/asmeta/download/AsmetaL_quickguide.html`.

[11] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[12] The NuSMV website. `http://nusmv.itc.it/`.

[13] The Eclipse website. `http://www.eclipse.org/`.

[14] Egon Börger. The Abstract State Machines Method for High-Level System Design and Analysis. Technical report, BCS Facs Seminar Series Book, 2007.

[15] Gerhard Schellhorn and Richard Banach. A concept-driven construction of the mondex protocol using three refinements. In *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 57–70, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] Mastercard international inc.: Mondex. `http://www.mondex.com/`.