Convention CIFRE 413 / 99 — LAAS-CNRS / Twam Informatique

Technical Report:

Model Checking and Object Orientation: A Tool Overview

François Taïani [1, 2]
Mario Paludetto [1]
Thierry Cros [2]

[1] Laboratoire d'Analyse et d'Architecture des Systèmes
LAAS-CNRS, 7 av. du Colonel Roche
F-31077 Toulouse CEDEX 4 / France
{francois.taiani, mario.paludetto}@laas.fr

[2] Twam Informatique
Les Espaces de Balma bâtiment 15
16 av. Ch. de Gaulle
F-31138 Balma CEDEX / France
{taiani, tcros}@twam.com

Abstract

Automatic program verification belongs to the old quests of software engineering, and theoretic computer science. Though any definitive solution is fundamentally excluded due to theoretical considerations, some partial yet promising answers could be proposed so far, in particular Model-Checking.

At the same time, Object Orientation has firmly established itself among practitioners as one of the most relevant programming paradigms.
In this report, we address the connection between Model-Checking and Object-Orientation from a tool perspective.

In a first part we try to give an overview of today's model checking state of the art with three representative tools among the most popular ones: SPIN, SMV and KRONOS. We introduce the spirits, the concepts, and the domain of application of each tool.

In a second part, we move on to higher-level frameworks: vUML, UMLAUT and Bandera. Those tools take profit of the "pure" model-checking engines presented before to allow the verification of Object Oriented Models. Among them, we identify 2 main approaches we refer respectively to as the preventive and the pragmatic one. We conclude with some considerations about the practical use of Model Checking methods.

1 Document Content

2 Introduction

The ability to check software and more generally systems against known requirements is a corner stone of any sound system engineering practice. In this context, the use of formal verification methods has been studied for several decades now as a promising mean to improve software quality. More particularly in the field of reactive systems, formal methods have been successfully applied to detect insidious flaws in a variety of complex systems, from communication protocols, to embedded systems, cryptography, and hardware design.

Model Checking [Merz00] and Theorem Proving [Rush00] are the two main approaches for the formal verification of discrete reactive systems in use today. In both cases, a model of the system is built, commonly as a discrete state / transition system. At this stage, the whole state / transition structure of the system is usually not explicitly expressed in the model. Rather, It is implicitly present through the description of each of its concurrent components. Besides the very description of the model, an ad hoc formalism allows the modeler to express properties on the model. The differences between Theorem Proving and Model Checking mainly lie in the automation degree of the verification process as well as in the nature of the properties that can be checked.

Theorem Provers usually cooperate interactively with the user to help build a mathematical proof of the considered property. Though success is not guarantied, a large range of properties may be considered and infinite systems are inherently allowed by the method.

Model Checking on the other hand is restricted in its classical form to finite state systems, and decidable logics. Model Checkers are normally fully automated, and given unlimited memory space and time necessarily produce a result. The practical limitation of model checking comes from the well-known state explosion problem, i.e. the exponential growth of the state space of a distributed model with respect to the number of its concurrent components.

A variety of tools have been implemented for Model Checking since the early days of formal verification, some of them going back to the beginning of the 80s. The research in this field has profited by the attention of a large community, who constantly improves the theoretical framework and the available tool set. In this report we try to give an overview of some the model checkers freely available today, in particular in connection with object oriented concepts. Naturally, the number of implemented model checkers would have been too large for our study. Because of our particular focus, we’ve chosen to present in a first time some of the best-known model checkers (Section 3: “Pure” Model-Checking). In a second time, we investigate some of the approaches combining object-orientation and model checking (Section 4: Connecting OO and Model-Checking). We then conclude on the general evolution of object-oriented model-checking approaches.

3 “Pure” Model-Checking

3.1 Introduction

In this section we are interested in Model-Checking in its “pure” understanding, as opposed to multi-level approaches, that we discuss in a second section (4 Connecting OO and Model-Checking). Of course, one can hardly speak of any “pure” Model-Checking, since a wide variety of techniques and approaches have been proposed and keep being proposed to the model-checking problem. We focus here on Model-Checking engines, which target performance and scalability. In this sense, these tools are more concerned with state-space compression and on-the-fly traversals that with ergonomics and conviviality. As such they may be qualified as “pure” Model-Checkers.

For this part, we have selected three model checkers among the most popular. Each of them exemplifies a certain philosophy, and spirit. The first one, SPIN, certainly the one with the longest history, specifically targets the verification of distributed software. The second one we look at, SMV, is more focused on hardware design, and was one of the first model checkers to introduce bridges to some theorem proving concepts. The last one, KRONOS, is a representative of dense time model checkers.

3.2 SPIN

3.2.1 Introduction

SPIN [Holz97] is a model checker that has been developed at the Computer Science Research Center of the Bell Laboratories. Historically SPIN goes back to the beginning of the 80's when the Bell Labs Formal Method Group started working on automatic protocol verification. Since then the tool has been constantly improved and extended with the leading model-checking technologies. SPIN uses the modeling language PROMELA (Process Meta Language) for the model description. PROMELA is an abstracted process description language, which focuses on inter-process communication. In particular PROMELA allows asynchronous process modeling.

According to the SPIN Web-Site at the Bell Laboratories, the following features are the distinctive characteristics of SPIN compared to other model checkers: [*Spin00]

- SPIN explicitly targets software verification.
  In particular the PROMELA language directly supports standard process synchronization constructs like rendezvous, buffered-message passing, and shared memory. SPIN also allows dynamic process creation and destruction.
- SPIN uses an on-the-fly state-space construction. Only the state-subspace relevant to the considered property is explored.
- SPIN fully integrates the Linear Temporal Logic (LTL) as formalism for input requirement.
- SPIN offers both exhaustive and partial proof techniques.
- SPIN uses partial order and optionally BDD-like storage for state-space reduction.

[Holz97] gives a very good overview of the tool architecture and functionalities. In the remainder of this section we will only give a basic introduction to the SPIN concepts, and spirit. Interested readers are referred to that first article for more details.

3.2.2 Processes

The first step when using SPIN consists in building a PROMELA model of the system one wants to check. The building bricks of a PROMELA description are processes and process types. Processes are instantiations of process types in the same way objects are instantiations of classes. (Though no object-oriented features like inheritance and polymorphism are included in PROMELA.)

```
proctype MyProcessType()
{
  bit internalState = 0;
  internalState = 1;
  printf("myInternalState:%d\n",internalState);
}
```

Figure 1: A simple process type definition in PROMELA

Figure 1 shows the definition of a simple process type in PROMELA. Instantiations of this process type have one local variable **`internalState`**, which is initialized to 0 and then set to 1. The process then terminates. For a PROMELA model to be executable, SPIN needs some entry point where some processes are instantiated. This is done in a default process called **`init`**, which is something like the **`main`** procedure of a C program.

```
proctype MyProcessType()
{
  bit internalState = 0;
  internalState = 1;
  printf("myInternalState:%d\n",internalState);
}

init
{
  run MyProcessType()
}
```

Figure 2: A small executable PROMELA model

Figure 2 shows a complete PROMELA model using the Process Type of Figure 1. The **`run`** keyword in the **`init`** process is used to create a running instance of **`MyProcessType`**. When run in simulation modus on this model, SPIN produces the following output (warnings and some minor output were omitted, process names are highlighted):

```
  0: proc - (:root:)       creates proc 0 (:init:)
  1: proc 0 (:init:)       creates proc 1 (MyProcessType)
  1: proc 0 (:init:)       line 10 "pan_in" (state 1) [(run MyProcessType())]
  2: proc 1 (MyProcessType) line  4 "pan_in" (state 1) [internalState = 1]
  3: proc 1 (MyProcessType) line  5 "pan_in" (state 2) [printf(...)]
  3: proc 1 (MyProcessType) terminates
  3: proc 0 (:init:)       terminates
2 processes created
```

Figure 3: Simulation Output of SPIN for the Model of Figure 2.

3.2.3 Channels

The communication paradigms of Promela are inspired from Hoare's CSP language [Hoare85]. As in CSP, PROMELA Processes can communicate through channels. A PROMELA channel is primarily a typed FIFO list. By default, any process that can see the channel (because the channel is global, or because it was passed as a parameter) may concurrently either write or read on the channel.

```
#define SIZE_OF_CHANNEL 10

chan channelFromInitToMyProcess = [SIZE_OF_CHANNEL] of {bit}
```

Figure 4: Example of a Channel Definition

Figure 4 shows the declaration of a channel **`channelFromInitToMyProcess`**, with a maximal capacity of 10 messages of type **`bit`**. This means that up to 10 messages containing a bit value each may be buffered in that channel. If **`channelFromInitToMyProcess`** is defined as a global channel, any process may write **`bit`** values to it with the **`!`** operator.

```
init
{
  run MyProcessType();
  channelFromInitToMyProcess!0
}
```

Figure 5: Init *Process writes a 0 to channel* channelFromInitToMyProcess

Figure 5 gives an example of the **`init`** process writing a **`0`** to the channel **`channelFromInitToMyProcess`**. In case the channel is full, writing to the channel is a blocking instruction: the writing process is blocked until some channel cells are freed through a symmetric read operation. A read is done with the **?** operator.

```
proctype MyProcessType()
{
  bit internalState = 0;
  channelFromInitToMyProcess?0;
  internalState = 1;
  printf("myInternalState:%d\n",internalState)
}
```

Figure 6: Instances of MyProcessType *wait for a 0 value on the channel* channelFromInitToMyProcess *before proceeding to* internalState *1.*

Figure 6 shows an example of a read operation on channel **`channelFromInitToMyProcess`**. Similarly to write operations, a process reading from an empty channel is blocked until a message is sent on the channel. Figure 6 put a further constraint on the read operation: **`channelFromInitToMyProcess?0`** requires the read message to contain the **`bit`** value **`0`**. If a **`1`** is the next value to be read instead, then the **`MyProcessType`** instance is blocked and can't proceed further. The **`1`** value has to be consumed by an other process for the channel **`channelFromInitToMyProcess`** not to be blocked itself.

If we put the pieces presented in Figure 4, Figure 5, and Figure 6 together, we get the executable PROMELA model of Figure 7.

```
#define SIZE_OF_CHANNEL 10

chan channelFromInitToMyProcess = [SIZE_OF_CHANNEL] of {bit}

proctype MyProcessType()
{
  bit internalState = 0;
  channelFromInitToMyProcess?0;
  internalState = 1;
  printf("myInternalState:%d\n",internalState)
}

init
{
  run MyProcessType();
  channelFromInitToMyProcess!0
}
```

Figure 7: An instance of MyProcessType *receives a 0 value through* channelFromInitToMyProcess

After a simulation run within the graphical environment XSPIN of the SPIN distribution, the model of Figure 7 generates the following output (the abbreviations `p.` for `proc` and `st.` for `state` were used for place reasons):

```
  0: p.- (:root:)        creates proc 0 (:init:)
  1: p.0 (:init:)        creates proc 1 (MyProcessType)
  1: p.0 (:init:)        line 15 "pan_in" (st.1) [(run MyProcessType())]
  2: p.0 (:init:)        line 16 "pan_in" (st.-) [values: 1!0]
  2: p.0 (:init:)        line 16 "pan_in" (st.2) [channelFromInitToMyProcess!0]
  3: p.1 (MyProcessType) line  8 "pan_in" (st.-) [values: 1?0]
  3: p.1 (MyProcessType) line  8 "pan_in" (st.1) [channelFromInitToMyProcess?0]
  4: p.1 (MyProcessType) line  9 "pan_in" (st.2) [internalState = 1]
  5: p.1 (MyProcessType) line 10 "pan_in" (st.3) [printf(...)]
  5: p.1 (MyProcessType) terminates
  5: p.0 (:init:)        terminates
2 processes created
```

Figure 8: Output generated from the Model of Figure 7

XSPIN also generates the following message sequence chart, showing message communication and process creation and termination. (See Figure 9.)

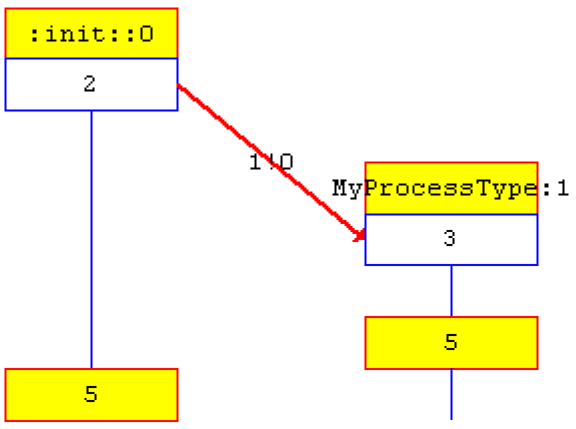


Figure 9: Message Sequence Chart Generated by XSPIN from the Model presented in Figure 7.

3.2.4 Indeterminism

In this small example, there is only one instance of **`MyProcessType`** and thus no conflict between different processes with respect to a unique channel resource. In the general case though, several processes may write and read concurrently to and from a channel. Because of that they may come in an access conflict situation. In the PROMELA semantics, such conflicts are random choice points: Any evolution of the system in which one of the conflicting process is selected for proceeding is a legal evolution. Of course, in a given simulation run, only one particular evolution among the many possible ones can be explored. Nevertheless, when checking a property, all possible behaviors relevant for the considered property are taken into account.

Channel conflicts are not the only possible source of indeterminism in a PROMELA model. Shared variables may be as well. Indeterminism may even be explicitly built in a model with dedicated constructs inspired from Dijkstra's guarded command language. This ability to model indeterminism is a central feature of PROMELA, since it allows the modeling of unpredictable or little known environments (faulty components, delayed channels, unknown task scheduling policy, unknown user behavior, and so on), which is essential in order to test the robustness of a system.

3.2.5 Loops

Another important feature is the possibility to model infinite loops in the model. Actually, most systems concerned by PROMELA can be regarded as infinitely cyclic systems, once abstracted from the specific start and termination phases.

```
init
{
  run MyProcessType();
  do
    :: channelFromInitToMyProcess!0
    :: channelFromInitToMyProcess!1
  od;
}
```

Figure 10: A non-deterministic cyclic version of the init *process*

Figure 10 shows a new version of the **`init`** process, in which infinitely many **`0`**s and **`1`**s are randomly written on the channel **`channelFromInitToMyProcess`**. The infinite loop is realized with the **`do...od;`** construct, in which each alternative begins with **`::`**.

```
proctype MyProcessType()
{
  bit internalState = 0;
  do
    :: channelFromInitToMyProcess?0;
       internalState = 1;
    :: channelFromInitToMyProcess?1;
  od;
}
```

Figure 11: the cyclic version of MyProcessType *matching the* init *process of Figure 10.*

Figure 11 presents the cyclic version of **`MyProcessType`** matching the cyclic production of **`0`**s and **`1`**s by the **`init`** process of Figure 10. This time the **`do..od;`** loop contains no indeterminism since the next message in the channel is either a **`0`** or a **`1`** but in no case both simultaneously. (Note that both alternatives may be blocked at the same time if the channel is empty.)

3.2.6 Model Checking with SPIN

If we let a simulation run randomly with the model of Figure 10, and Figure 11, we may get a message sequence chart like the one on Figure 12. Such a simulation usually never ends. When a system with cyclic components actually ends, the particular end-state most of the time corresponds to a deadlock. SPIN directly supports search for deadlocks, by checking if any global system state may be reached from which no further evolution is possible. Except for particular cases a deadlock indicates a system flaw.

Another type of relevant faulty behaviors are live-locks, in which the system get caught in some infinite idle loop. Of course, SPIN can't differentiate live-locks from normal cyclic runs without any further information about the model. This extra information is added to the model with the **`progress`** keyword, which should label process states in which the system "actually does something". From a model annotated with **`progress`** labels, SPIN is able to identify system loops with no progress states, which correspond to live-locks.

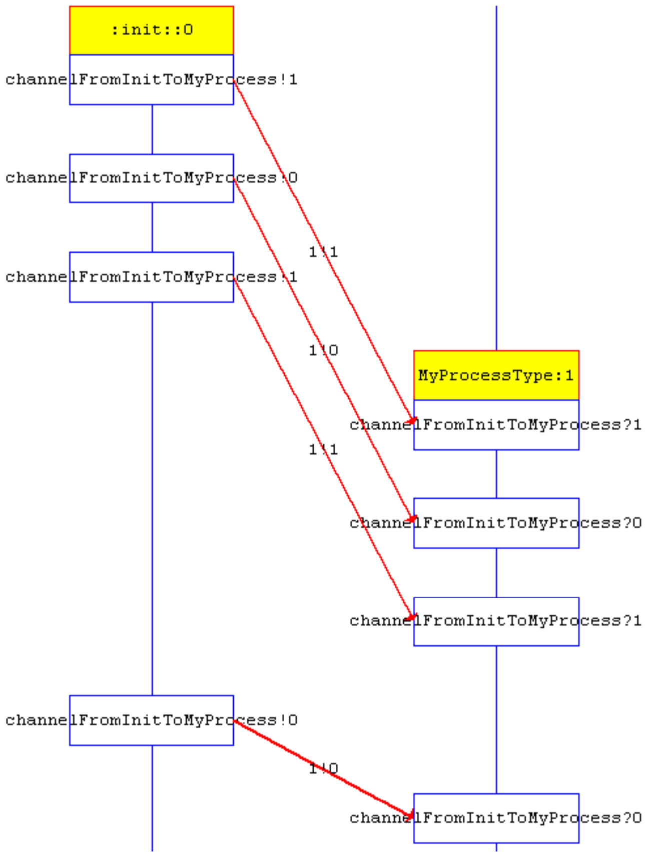

:init::0
channelFromInitToMyProcess!1
channelFromInitToMyProcess!0
channelFromInitToMyProcess!1
1!1
1!0
MyProcessType:1
channelFromInitToMyProcess?1
1!1
channelFromInitToMyProcess?0
channelFromInitToMyProcess?1
channelFromInitToMyProcess!0
1!0
channelFromInitToMyProcess?0

Figure 12: Random simulation with the model from Figure 10 and Figure 11

Dead-locks and live-locks are standard flaws one is interested in when verifying a model. SPIN further allows the user to specify more specific requirements in Linear Temporal Logic. This logic permits building formulae over runs of the system. Given a LTL formula and a run, either the run fulfills the formula or not. A system verifies a LTL formula if all its possible runs verify the formula themselves. The basic building bricks of an LTL formula are atomic state propositions. For example **`(internalState==0)`** characterizes all system states in which the variable **`internalState`** equals **`0`**. LTL further offers the quantifiers “eventually” which is noted **<>** in SPIN, and its dual form “always”, which in noted **`[]`**. For example, a requirement that every run of the system should eventually lead to a state in which **`internalState`** is equal to **`1`** is expressed by:

```
<>(internalState==1)
```

The requirement that within every run, **`internalState`** should be infinitely often equals to **`0`** and infinitely often equals to **`1`** is expressed by (**&&** means and):

```
[] ( (<>(internalState==1)) && (<>(internalState==0)) )
```

If we test the above property on the PROMELA model presented in Figure 10 and Figure 11, we get a counter example violating the property (Figure 13), in which **`init`** only produces **`1`**, and thus **`internalState`** always stays at **`0`**.

```
 2: p. 0 (:init:)        line 17 "pan_in" (st.1)    [(run MyProcessType())]
 4: p. 0 (:init:)        line 20 "pan_in" (st.-)    [values: 1!1]
 4: p. 0 (:init:)        line 20 "pan_in" (st.3)    [channelFromInitToMyProcess!1]
<<<<<START OF CYCLE>>>>>
 6: p. 0 (:init:)        line 20 "pan_in" (st.-)    [values: 1!1]
 6: p. 0 (:init:)        line 20 "pan_in" (st.3)    [channelFromInitToMyProcess!1]
 8: p. 0 (MyProcessType) line 11 "pan_in" (st.-)    [values: 1?1]
 8: p. 0 (MyProcessType) line 11 "pan_in" (st.3)    [channelFromInitToMyProcess?1]
```

Figure 13: Counter example for formula [] ((<>(internalState==1)) && (<>(internalState==0)))

3.2.7 Distribution, Reusability

The present introduction to SPIN is based on the Version 3.3.10 of the SPIN distribution. SPIN is available on UNIX, Windows 95/98 and NT. SPIN itself comes in a command line version, which generates ANSI-C code from a PROMELA model. The C code generated implements a dedicated program which depending on the options passed to SPIN either simulates or checks the model against a certain property. Once compiled this C program returns its results in a plain ASCII format.

Because of this, SPIN can quite easily be integrated in a more general framework, to serve as a powerful model checking engine. vUML and Bandera, which are discussed later in this document, take advantage of SPIN in this way.

The SPIN distribution also contains a graphical user interface programmed with Tcl/Tk, called XSPIN. This user interface offers all the functionalities supported by the SPIN command line version within a multi-windows application. It also generates graphical Message Sequence Charts from simulation runs and counter examples.

SPIN requires a C compiler and a C preprocessor to run correctly. XSPIN requires Tcl/Tk to be installed as well. SPIN is freely available for research and educational use. For commercial use, a small administrative fee is requested.

3.2.8 Conclusion

In this section we have given a basic introduction to the model checker SPIN and the modeling language PROMELA. SPIN models are built around the root concepts of processes and channels. SPIN allows the user to simulate models to gain a first understanding of its behavior and already offers advanced output formats like graphical message sequence charts. From a given model and any requirement expressed in a Linear Temporal Logic Formula, and provided with unlimited memory and time, SPIN is always able to check the model against the formula.

This short presentation could only address the most elementary features of SPIN. In particular partial verification techniques have not been exposed. Interested readers are referred to [Holz97] for further details.

3.3 Symbolic Model Verifier (SMV)

3.3.1 Introduction

SMV is a model checker first developed by Ken McMillan during his Ph.D. Thesis at the Carnegie Mellon University. SMV illustrates the central ideas McMillan investigated in his thesis, which was published in 1992. Since then the original SMV system has been split into 2 different versions. McMillan further develops the first one after he has moved to the Cadence Berkeley Laboratories in California. The results of his research are used by the Cadence Company for its verification product line for hardware design. The second version of SMV is maintained by the Model Checking Project Group of the Carnegie Mellon University, where McMillan originally implemented the first SMV program.

At Cadence, McMillan has specifically focused the further development of SMV on the verification of synchronous integrated circuits. In the meanwhile, at CMU, SMV is used as an experimental framework for general model checking. As a consequence, the 2 versions diverge in numerous essential points. In particular the language constructs and the logic flavors they use are not the same. The Cadence SMV Version has been specialized to use only LTL (Linear Temporal Logic) formulae to express properties on execution paths, while the CMU SMV Version uses CTL* (Extended Computation Tree Logic). In the same manner the **`process`** keyword, which permits the asynchronous composition of parallel state machines, was suppressed from the Cadence SMV Version. On the other hand the Cadence SMV version was extended with central constructs for model refinement (**`layer`** keyword), and compositional verification. In this report we specifically focus on the Cadence Version of SMV, which is the best documented of both. (See [McMill92] and [*McMill99] for a detailed description of both systems.) In the remainder on the text, SMV, when not stated differently, thus exclusively refers to the SMV Cadence Distribution in the experimental release 12-09-99.

3.3.2 Binary Decision Diagrams

The SMV Cadence distribution is based on the original CMU SMV release, and thus takes over its most essential features. One of them and actually the root idea of the original SMV is to use Binary Decision Diagrams (BDDs) [Bryant86] to code finite state-transition systems ([BCM92] and [McMill92]). BDDs are a compact representation of boolean functions, and as such they can be used to store sets of states and transition relationships. A BDD representation is said to be symbolic, because it does not extensively list all elements of a set (resp. all pairs of a relationship). This approach has proved to be very interesting for integrated circuits, because these circuits show strong structural regularities, which can be very efficiently stored by BDDs.

The use of BDDs in SMV is important to understand the way the system performance depends on the chosen model specification. Yet SMV uses BDDs in a totally transparent way from the user point of view.

3.3.3 SMV Basics

Like SPIN, SMV comes with its own modeling language, the SMV language [*McMill99]. This language is a synchronous modeling language built around the signal concept and the **`next`** operator.

```
typedef INTEG 0..255;

module main() {
  x: INTEG ;
  init(x) := 0 ;
  next(x) := x+1 ;
}
```

Figure 14: Basic SMV Model

For example the SMV Model of Figure 14 defines one signal **`x`** of type **`INTEG`**. **`x`** is initialized with the value **`0`**, and evolves according to the equation **`next(x) := x+1`**. A simulation of this model with SMV delivers the following trace:

```
x : 0 1 2 3 4 5 6 ..
```

Figure 15: Simulation of the Model of Figure 14

Properties to be verified are stated using the **`assert`** keyword. As in SPIN, SMV properties are stated using the Linear Temporal Logic (LTL). In SMV the *always* quantifier is written **`G`** for *globally* and the *eventually* quantifier is written **`F`** for *finally*. For example the property presented in Figure 16 would have been written **`[](<>(x==255))`** using the SPIN notation. Figure 17 gives the complete SMV model with the property.

```
propertyToCheck : assert G(F x=255);
```

Figure 16: variable x infinitely often reaches the value 255 in the model of Figure 14

```
typedef INTEG 0..255;

module main() {
  x: INTEG ;
  init(x) :=0 ;
  next(x) := x+1 ;
  propertyToCheck : assert G(F x=255);
}
```

Figure 17: Complete SMV Model with property

3.3.4 Induction

Besides BDDs, induction techniques are another important contribution of the original SMV that is present in the Cadence Version. Induction is particularly used in SMV to check systems with an arbitrary number of components. This approach was one of the first steps to combine core Model Checking with some Theorem Proving and has been pursued since then as a promising research direction [Rush00].

In SMV properties are identified by names, so that intermediate lemmas may be used to progressively build a proof.

```
ordset TYPE 0..255;
module main() {
  x: INTEG ;
  init(x) :=0 ;
  next(x) := x+1 ;

  forall (i in TYPE)
    p[i]: assert F(x=i);

  forall(i in TYPE)
    using p[i] prove p[i-1];
}
```

Figure 18: Simple Proof using induction and lemmas

Figure 18 is an example taken from [McM99b, p.40], which illustrates the use of lemmas within SMV. In this particular case the construct **`using property1 prove property2`** is used to prove a property by induction. The property targeted by the model of Figure 18 conjectures that the signal **`x`** finally reaches the value **`255`**: **`F(x=255)`**. To come to that result the first **`forall`** construct in the model potentially defines 255 properties **`p[0], p[1],...,p[255],`** which are then recursively checked in the second **`forall`** construct.

Things don't really occur that way in this particular example, because the type **`TYPE`** is actually defined as an ordered set (**`ordset`** keyword). This restricts the legal operations on such a set to incrementation, decrementation and comparison to the limits (here **`0`** and **`255`**) (see [McM99b, p.39] for more details). This information on the model allows SMV to reduce the set of properties to be proven to **`p[0], p[1], p[2],`** and **`p[255]`**. (We don't address here the justification of such a reduction. Interested readers are referred to the work of McMillan on that topic in particular to Chapter 5 of [McMill92].)

3.3.5 Layers, Abstractions, and Refinements

As mentioned above, another core feature of Cadence SMV is to allow several refinement layers of description. Higher layers offer an abstracted description of the system. They build its specification, and describe what it *should* do. The lower layers implement the higher layers, and describe *how* the system actually does what it should do. When chosen judiciously, the specification layers can drastically reduce the state space of a model by locally simplifying its description.

The verification of a system against a given property is done is two phases when using refinements: first the coherence between specification and refinement is checked. If both are coherent with each other the property is checked on the abstracted system. This abstracted system is usually considerably smaller than the original one, so that in numerous practical cases this two phase approach proves to be globally more efficient than a direct model checking. [McMill97]

Except for the following discussion in section 3.3.6, we don't enter in the details of the refinement technique in that report for place reasons. A good introduction to it can be found in [*McM99b]. This technique is particularly well suited to hardware design, because most low level circuit architecture contains intermediary storage units (register, carries, and so on). These elements are essential to the work of a circuit, but can be abstracted away when considering the circuits from a sheer functional point of view. Eliminating those auxiliary variables from the model into a specification layer considerably reduces the state space. Of course, only properties that don't directly address the detailed implementation of the circuit can then be checked on the resulting model. Besides a state-space reduction, another central advantage of this approach is its compositional aspect. Actually, once a given specification layer has been proved to fulfill a given requirement, any implementation respecting that specification will respect it as well.

3.3.6 Refinements and Software Verification

Nevertheless in the case of software verification, it's not clear in which extend the SVM refinement technique as present in the tool can improve program verification. Redundant data storage is not as typical in software as in hardware, which notably limits the range of the technique. Additionally, in its actual version (12-09-99), SMV requires refinements to map bijectionnally. As a consequence, the specification layer can't describe a broader set of acceptable behaviors than those implied by the implementation layer. In particular the specification layer can't approximate the implementation by introducing indeterminism.

```
typedef SIMPLE_INT 0..7;

module main() {
  highY, lowY: SIMPLE_INT ;
  carry: Boolean;

  init(highY):=0;
  init(lowY) :=0;
  init(carry):=0;

  if (!carry) {
    if (lowY<7) {
      next(lowY):=lowY+1;
    }
    else {
      next(carry):=1;
    }
  } /* end if not carry */
  else {
    if (highY<7) {
      next(lowY) := 0;
      next(highY):= highY+1;
      next(carry):= 0;
    }
  } /* end if carry */

}
```

Figure 19: A 2-bytes counter with one carry

For example let's consider the model of Figure 19. This example describes a counter stored in two 3-bit variables (**highY** and **lowY**), which is incremented from 0 to 63, and then remains at 63. A normal incrementation takes one time unit, except when a carry occurs on the low weight byte **lowY**, when it take 2 time units, because of the carry operation.

```
lowY :  0 1 2 3 4 5 7 7 0 1 2 3 4 5 6 7 7 0 ...
carry:  0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 ...
highY:  0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 ...
```

Figure 20: In the model of Figure 19, a carry operation take 2 time units.

Suppose we want to abstract that model in an 8-bit-counter, one possibility could be to use the following model as an abstraction layer: (Note the use of an extra module **eightBitCounter** for modularity.)

```
typedef DOUBLE_INT 0..63;

module eightBitCounter() {
  y      : DOUBLE_INT ;
  random : boolean ;

  init(y)     :=0;

  if (y<63) {
    random := {0,1};
    switch(random) {
      0 : next(y):=y+1;
      1 : next(y):=y;
    } /* end switch(random) */
  } /* end if (y<63) */
}

module main() {
  counter : eightBitCounter;
}
```

Figure 21: A proposed abstraction for the model of Figure 19

This model is non-deterministic and accepts any evolution of **`y`** in which **`y`** steadily grows up to 63. Indeterminism is introduced at the line **`random := {0,1}`**, where **`0`** or **`1`** is randomly allocated to signal **`random`**. Note that **`y`** may in particular stays constant. We want to check if the implementation of Figure 20 is actually coherent with this specification. To do so, we simply add the declaration of module **`eightBitCounter`** to Figure 20, along with the instantiation of a that module in the main module, and the following **`layer`** declaration[1]:

```
layer specification : {
  highY := counter.y / 8;
  lowY  := counter.y mod 8;
}

prove highY//specification ;
prove  lowY//specification ;
```

Figure 22: layer construct for refinement mapping

The **`layer`** declaration defines the mapping between **`counter.y`** and **`(highY,lowY)`**.Unfortunately, SMV will fail to prove the relationships of the layer **`specification`**, because the module **`counter`** allows behaviors for **`counter.y`** (like for example staying constant) that are impossible for **`highY`** and **`lowY`**. Stated more generally, SMV does not permit to approximate components away (here the signal **`carry`**) by introducing indeterminism in a model. In this particular case it is further questionable if the proposed refinement would bring any space reduction, since **`carry`** is the only signal to be taken out through the **`eightBitCounter`** description. This is because we have only one carry. If we had lots of basic bit-adders with several carry signals to propagate overflows, eliminating the carries through a similar abstraction would certainly make more sense.

3.3.7 Distribution, Reusability

The Cadence Version of SMV is available on Internet from the personal pages of McMillan at the Berkeley University [*McMill00]. The download license grants a free use of Cadence SMV for internal research and development activities, but excludes any incorporation of SMV in a commercial product. SMV can primarily be used as a command line program but also comes along with a graphical interface programmed with Tcl/Tk. As such the integration of SMV in a more general development

[1] In SMV, the **`layer`** statement is a sort of 'glue' construct. It connects two levels of abstraction together by defining 'bridge' relationships between the values of each levels.

environment can be done quite easily. SMV is available on Windows NT, Windows 95, Sparc/Solaris, Sparc/SunOS, HPUX, MIPS/Irix and i386/Linux.

3.3.8 Conclusion

Cadence SMV is explicitly dedicated to hardware verification. Though, the ideas developed in the tool may be quite interesting for software verification, if not necessarily to the same extend. Actually, techniques that were first tested and validated within SMV like BDDs have been incorporated since then in other famous Model Checkers like SPIN. The direct support for refinement and abstraction by the modeling language and the introduction of some automatic proving techniques like induction are further distinctive features of SMV, which make it particularly interesting.

3.4 KRONOS

3.4.1 Introduction

KRONOS is a model checker dedicated to clock automata, i.e. automata with a dense time representation. KRONOS has been developed at the Verimag laboratory (Grenoble) around Sergio Yovine since the beginning of the 90s. As stated in [Yovine97]:

> *KRONOS [9, 10, 8, 11, 26, 20, 6, 5] is a software tool built with the aim of assisting designers of real-time systems to verify whether their designs meet the specified requirements. One major objective of KRONOS is to provide a verification engine to be integrated into design environments for real-time systems in a wide range of application areas. Real-time communication protocols [9, 10], timed asynchronous circuits [20, 6], and hybrid systems [23, 10] are some examples of application domains where KRONOS has already been used. KRONOS has been applied to analyze real-time systems modeled in several process description formalisms such as Atp [22], Aorta [7], Et-lotos [9], and T-argos [19]. Several projects are currently under development to connect KRONOS to specification languages such as Shift [1], Diadem [12], and Tces (Timed Condition/Event Systems) [17].*

KRONOS stays apart from the two previously presented model checkers SPIN and SMV in that it tackles models with dense time representation. Contrarily to SPIN and SMV, KRONOS does not provide any modular-structured modeling language, and directly works on an automaton level. KRONOS yet provides a certain modularity degree through its ability to compute the product of a timed automaton, with synchronization labels.

3.4.2 Timed Automata

Timed automata are obtained through the extension of normal state / transition automata with clock variables. Clock variables have real values, and all evolve at the same speed as "time passes". Clocks may be reset during a run as well, consequently to the firing of a transition.

The transitions of a timed automaton may be constrained by guard conditions on those clocks[2]. Semantically, such clock guards constrain the time date of the transition firing: Depending on the values of the clocks, a given transition may be enabled or not.

[2] Only some specific equations are allowed as guard conditions for reasons of computability. More precisely, only boolean combination of some specific atomic equations are legal. The comparison between a clock, and a rational constant ($\mathbf{x} <= \mathbf{K}$), and the comparison between a difference of clocks, and a rational constant ($\mathbf{x} - \mathbf{y} > \mathbf{K}$) are the allowed atomic equations.

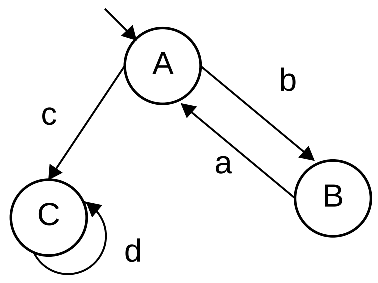


Figure 23: A simple normal automaton

For example the automaton in Figure 23 presents a plain automaton which accepts the following event sequences: **`cdddd..`, `bacdddd...`, `babacdddd...`**, etc.. Those sequences can be summed up into the following regular expression : $\mathbf{(ba)^{*}c(d)^{\omega}}$. Figure 24 is a timed version of the automaton of Figure 23. The acceptation of an event sequence by this automaton still depends on the events that make the sequence up: the sequence **`acac...`** for example is neither accepted by the automaton of Figure 23 nor by the one of Figure 24.

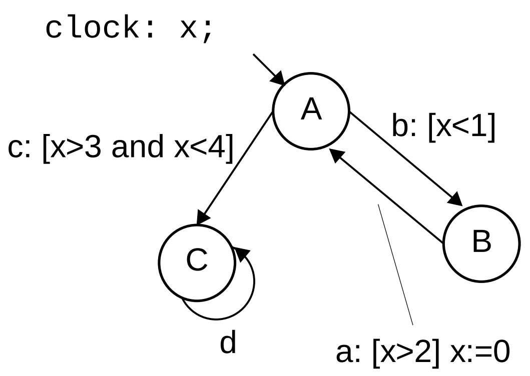


Figure 24: Automaton from Figure 23 extended with clocks

In the automaton of Figure 24, transitions **a**, **b**, and **c** are additionally constrained by conditions on the clock **`x`**. For example transition **b** cannot be fired after **`x`** has gone beyond **`1`**. Transition **a** must be fired after **`x`** has gone beyond 2, and resets the clock **`x`** to **`0`** when it is fired. The automaton itself initially starts in state **`A`** with clock **`x`** set to **`0`**.

As a consequence a sequence like **`babacddd..`** which is accepted by the Automaton of Figure 23 may or may not be accepted by the automaton of Figure 24, depending on the dates at which the events occur. To account for that extra criterion, events are represented with an extra time stamp corresponding to their occurrence date. For example the following timed version of **`babacddd..`** is accepted by the automaton of Figure 24:

(**b**,0.5)(**a**,3.2)(**b**,4.1)(**a**,14.1)(**c**,17.5)(**d**,18)(**d**,19)(**d**,20)...(**d**,n)...

The evolution of the clock **x** for that sequence would be the following:

```
States Transitions  Clock      Comment

A->B:  (b,  0.5)    x==  0.5   respects [x<1]
B->A:  (a,  3.2)    x==  3.2   respects [x>2]
                    x:=  0     x is reset
A->B:  (b,  4.1)    x==  0.9   respects [x<1]
B->A:  (a, 14.1)    x== 10.9   respects [x>2]
                    x:=  0     x is reset
A->C:  (c, 17.5)    x==  3.4   respects [x>3 and x<4]
C->C:  (d, 18.0)    x==  3.9   no constraints
C->C:  (d, 19.0)    x==  4.9   no constraints
...
```

3.4.3 TCTL Logic

The theory of timed automata was given a solid theoretical basis by Alur and Dill at the beginning of the 90's [Alur+94]. A special flavor of the CTL logic, the Timed Computation Tree Logic (TCTL), was developed for these automata [Alur+93]. TCTL allows explicit reference to time and the specification of duration between events. The most important result at the origin of that formalism is the following: The verification of a TCTL formula on a timed automaton is decidable and can be reduced to the verification of a CTL formula on a plain automaton [Alur+93].

Like CTL, TCTL is a state logic, contrarily to LTL, which is a path logic. It means that TCTL formulae are expressed on the states of a timed automaton. A state of a timed automaton is a pair containing a state of the underlying plain automaton (**A**, **B** or **C** in Figure 24), and a value vector for the clocks of the automaton. For example **(A,x:0.5)** and **(B,x:3.2)** are states of the timed automaton of Figure 24.

Atomic properties can be expressed on the clock values of a state: For example the property **(x<2)** is verified by the state**(A,x:0.5)**, which is noted **(A,x:0.5)**$\models$**(x<2)**, but not by the state**(B,x:3.2)**. The future evolution of the automaton from a particular state is expressed by the timed quantifier "certainly" $\forall\Diamond_{op.K}$ and the timed quantifier "possibly" $\exists\Diamond_{op.K}$ where **op.** is some comparison operator among **<**, **>**, **=**, **<=**, **>=** and **K** is some rational constant[3]. With these operators allows representing the necessity or the possibility of the occurrence of an event with respect to some time constraint. For example the formula on Figure 25 states that, every time the system is in state **A**, and the clock **x** equals **0**, then it's impossible to reach state **C** before **2** time units.

```
(current_state_is_A and x=0) => not (∃◇<2 current_state_is_C)
```

Figure 25: A TCTL property that is valid on all states of the automaton of Figure 24

3.4.4 KRONOS functionalities and techniques

We don't address in this report the actual input format of the KRONOS tool, as we have done for SPIN and SMV. This format translates the formalism of timed automata we've just presented, and the reader is referred to [Yovine97] for a complete presentation.

KRONOS has three main functionalities: computation of the product of several timed automata, checking of a TCTL formula on one automaton, and time abstracting reduction of one automaton.

The time abstracting reduction of a timed automaton takes time out of a time model to obtain a behavior-equivalent plain automaton (Time Abstracting Equivalence). Such an equivalence amounts to the elimination of explicit time representation in the model, but conserves the causality relationship between events. One should note that in most cases the plain automaton underlying a timed

[3] We have chosen to present the restricted version of TCTL proposed in [Yovine97]. In fact, as stated in [Yovine93, p. 32], this version is less expressive than the complete original TCTL proposed for example in [Alur+93]. Interested readers are referred to [Yovine93, pp.25] for more precisions.

automaton is not behavior-equivalent to this timed automaton. This is because time constraints induce extra interdependences between events that are not present in the original plain automaton. The resulting abstracted automaton can then be compared under various equivalences to a specification automaton. KRONOS does not check such equivalences itself, but relies instead on the tool Aldebaran of the INRIA, which is part on the CADP (Caesar / Aldebaran Development Package) tool set. (See for example [Gara+97] for an overview.)

As presented in [Bozg+98], KRONOS takes advantage of a wide range of technologies. Symbolic state representation, on-the-fly traversal, and abstractions are proposed to reduce the state space explosion. Here one should note the need to use difference bounds matrices (DBM) conjunctly with binary decision diagrams (BDD) for symbolic representation. We can't enter in the detail of that technique here, but the reader is referred to the thesis of S.Yovine [Yovine93] on that subject.

3.4.5 Distribution, Reusability, Similar Tools

KRONOS presents itself as a command line program and is available for Sun Solaris 5.7, Linux and Windows 98 / NT (State on August 31rst 2000). The tool Aldebaran, which is needed to exploit the time abstracting reduction feature is available on SunOS 4.* (Solaris 1.*), SunOS 5.* (Solaris 2.*), and Linux 2.0.

KRONOS is freely distributed for academic institutions for non-profit use, and can be downloaded from the VERIMAG site [*Kron00].

Last, but not least, the tool UPPAAL must be mentioned besides KRONOS. UPPAAL offers like KRONOS model-checking functionalities on timed automata. UPPAAL is jointly developed and promoted by the Design and Analysis of Real-Time Systems group at Uppsala University (Sweden) and by the Basic Research in Computer Science group at Aalborg University (Denmark). Like KRONOS, UPPAAL is freely available for non-profit research purposes [*Upp00]. In particular neither UPPAAL nor any part of its code should be used or modified for any commercial software product. In its present version (3.0.41), UPPAAL is available under a fully integrated Java graphical interface, contrarily to KRONOS, which only comes as a command line program.

3.4.6 Conclusion

The range of models addressed by KRONOS and UPPAAL and hence their expressiveness go far beyond those of SPIN and SMV. KRONOS and UPPAAL are specialized on dense time model checking, which clearly set them apart from standard model checking tools. Yet one should be conscious that the representation of dense time systems and their properties further worsen the state explosion problem. As a consequence the systems that KRONOS and UPPAAL can tackle are notably smaller than those of SPIN and SMV. Furthermore, the current version of KRONOS and UPPAAL don't offer any specific modeling formalism except timed automata, contrarily to SPIN and SMV. The absence of dedicated modeling languages in KRONOS and UPPAAL is certainly to be related with the weak support for explicit time representation in most structured programming and modeling languages.

4 Connecting OO and Model-Checking

4.1 Introduction

In the precedent section we have looked at some of the best known model checkers of today's verification field. Though some of them offer dedicated modeling languages (PROMELA in SPIN or the SMV language in SMV), those languages clearly remain restricted to some specific aspects of the systems they describe. They definitely don't intend to offer the structures and the flexibility of a fully enabled programming or modeling notation. This is not their purpose. This would even burden the verification task of the corresponding model checkers. Pure model checking tools assume the existence of an ad-hoc model of a system. They don't address the actual construction of that model.

On the other hand, the modeling and structuring of software has always been a critical issue in software engineering. In the comparatively long history of what we may call "software notation", Object

Orientation is one of the last paradigms to have firmly established itself in the industry. Because of that, and for some years now, several research teams have tried to combine Object Orientation with "pure" verification tools. In this section, we try to give a brief overview on that topic with three tools that have been proposed in that context. The first two tools, vUML and UMLAUT, are focused on the OO notation UML (Unified Modeling Language). The third one, Bandera, more specifically addresses the verification of Java programs.

4.2 vUML

4.2.1 Introduction

vUML [Lil1+99] is a transformational tool able to convert UML [*UML1.3] models into PROMELA descriptions, which can then be processed by SPIN [Holz97]. vUML is being developed at the Åbo University (Finland), under the direction of Professor Johan Lilius. The current prototype version (0.21) of vUML was implemented by Iván Porres Paltor. Here is how the official web-site of vUML describes the tool [*vUML00]:

vUML is a tool that automatically verifies UML models. [...] vUML verifies models where the behaviour of the objects is described using UML Statecharts diagrams. It supports concurrent and distributed models containing active objects and synchronous and asynchronous communication between objects. The tool uses SPIN model checker to perform the verification, but the user does not have to know how to use SPIN or the PROMELA language. If an error is found during the verification, the tool creates a UML sequence diagram showing how to reproduce the error in the UML model.

(source http://www.abo.fi/~iporres/vUML/, state as on 16th August 2000)

In the remainder of this section we introduce vUML based on the example of the dining philosopher problem presented in [Lil1+99]. The code samples are adapted from the corresponding program provided with the public distribution of vUML [*vUML00].

4.2.2 Representation of UML-Models with vUML

vUML is programmed in Python [*Pyth00] [AscLu99], an interpreted OO programming language. The models to be checked by vUML are described as Python programs as well, using the UML meta-model implemented by the tool. For instance Figure 26 presents how the fork object of the dining philosopher problem is represented in the sample of the vUML distribution:

```
eGet     = CallEvent("get")
eRelease = CallEvent("release")

Fork = UMLClass(
        name      = "Fork",
        behaviour = StateMachine(
                       name        = "ForkBehaviour",
                       top         = Main,
                       states      = [Available,Taken],
                       transitions = [
                          Transition(
                            name    = "tGet"    , source = Available,
                            trigger = eGet      , target = Taken
                          ),
                          Transition(
                            name    = "tRelease", source = Taken,
                            trigger = eRelease  , target = Available
                          )
                       ] # End transitions
                     ) # End StateMachine
    ) # End UMLClass
```

Figure 26: Description of a fork class using the UML meta-model of vUML (incomplete)

In Figure 26 the elements of the UML Meta-Model are bold typed, whereas their attributes are represented in italic. Note the particular Python syntax, which allows the explicit reference to formal parameter names. The State Machine coded in Figure 26 is shown in Figure 27 with the usual UML graphical notation.

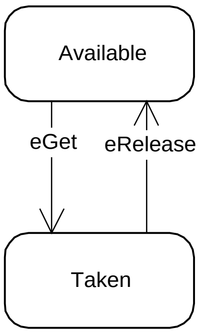


Figure 27: State Chart Diagram of the class ***`Fork`*** *presented in Figure 26 (source [Lil1+99], p.3)*

4.2.3 Semantics of UML State Machines

Unfortunately, the action semantics of UML is not precise enough to allow an unambiguous translation from an UML state machine into a particular formal notation. Because of that, Lilius and Paltor had to settle for a given semantics. The semantics they chose is described in [Lil2+99]. The core of that semantics consists in the explicit description of the Run To Completion (RTC) step introduced in [*UML1.3, pp. 2-149], as well as a precise handling of transition conflicts and deferrable events.

```
Philosopher =
   UMLClass("Philosopher",
      attributes = None,
      behaviour  = StateMachine(
                      name        = "philosopherStateMachine",
                      top         = PhM,
                      states      = [thinking,eating,wfr,dropl],
                      transitions= [
                          Transition(
                             name    = "gLeft",
                             trigger = eAUTO,
                             source  = thinking,
                             target  = wfr,
                             effect  = Action(send=Send(left,eGet))
                          ),
                          Transition( [...] ),
                          Transition( [...] )
                          Transition( [...] )
                      ]
                  ),
      associations=[left,right]
   )
```

Figure 28: An example of object communication with vUML: a Philosopher sends an **`eGet`** *event to its left fork* **`left`** *when processing transition* **`gLeft`**.

Lilius and Paltor also chose PROMELA to express the actions and the guards of UML transitions and the activities of UML states. In Lilius and Paltor's semantics, objects communicate along static association instances (links), which are considered as delayed reliable channels: events may be arbitrarily delayed, but can't be lost. Figure 28 shows an example of such an object communication taken from the dinning philosopher sample of the vUML distribution: Instances of the UML class **`Philosopher`** send instances of the event **`eGet`** down their **`left`** link. This link leads in turn to a **`fork`** object. (Figure 26)

4.2.4 Translation to a PROMELA Program

In order to be processed by vUML, a model must contain a collaboration diagram. This diagram instantiates objects and links based on the UML classes previously described in the model. As such the collaboration diagram sets up the runable system that will actually be checked by SPIN.

```
dinner = CollaborationDiagram(
            name      ="dinner",
            instances = [
              Instance("John" ,Philosopher) , Instance("Fork1",Fork),
              Instance("Anna" ,Philosopher) , Instance("Fork2",Fork),
              Instance("Peter",Philosopher) , Instance("Fork3",Fork)
            ],
            links = [
              Link("John" ,"Fork1", None, left ),
              Link("John" ,"Fork3", None, right),
              Link("Anna" ,"Fork2", None, left ),
              Link("Anna" ,"Fork1", None, right),
              Link("Peter","Fork3", None, left ),
              Link("Peter","Fork2", None, right)
            ]
         )
```

Figure 29: A collaboration diagram instantiating a Dining Philosopher Problem.

Figure 29 shows such a collaboration diagram using the Philosopher and Fork classes presented in Figure 26 and Figure 28. The corresponding graphical UML representation of that diagram is shown in Figure 30. Note that contrarily to usual collaboration diagrams, those used in vUML don’t contain any interaction [*UML1.3, Part 2.10 Collaborations, pp. 2-103], i.e. any explicit message exchanges. This is because they don’t represent any particular scenario, but instead instantiate a particular context in which model checking can take place.

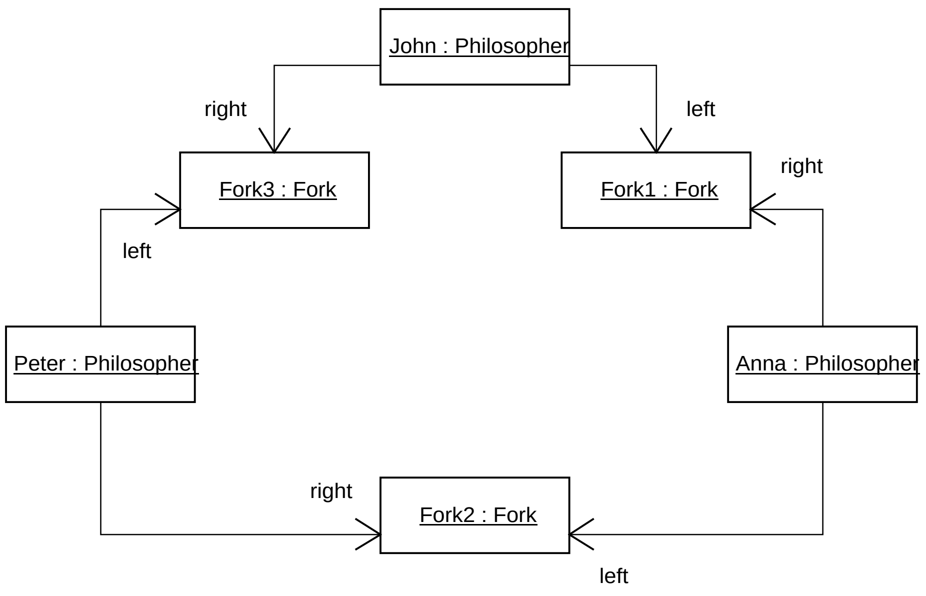
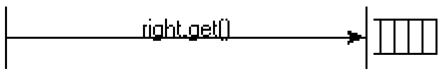


Figure 30: Graphical representation of the collaboration diagram defined in Figure 29 (source [Lil1+99, p.4])

Based on the UML model provided to it, vUML maps each UML class to a PROMELA process type (**`proctype`** keyword). Those process types are then instantiated in the **`init`** process of the PROMELA program accordingly to the collaboration diagram of the UML model. Links are implemented using PROMELA channels (**`chan`** keyword).

4.2.5 Results returned by vUML

After the PROMELA program has been generated, vUML launches a deadlock search using SPIN. As stated in [Lil1+99], invalid states, live locks, constraint violations and various queue overflows are checked as well. Figure 31 shows the sequence diagram returned by vUML for the collaboration diagram of Figure 29. Note that the deferring of the deferrable call event **`eGet`** is represented with an arrow leading to a queue symbol:



In this example, the sequence diagram of Figure 31 ends on a deadlock situation, in which each philosopher has taken its left fork and waits for his right neighbor to release its right fork.

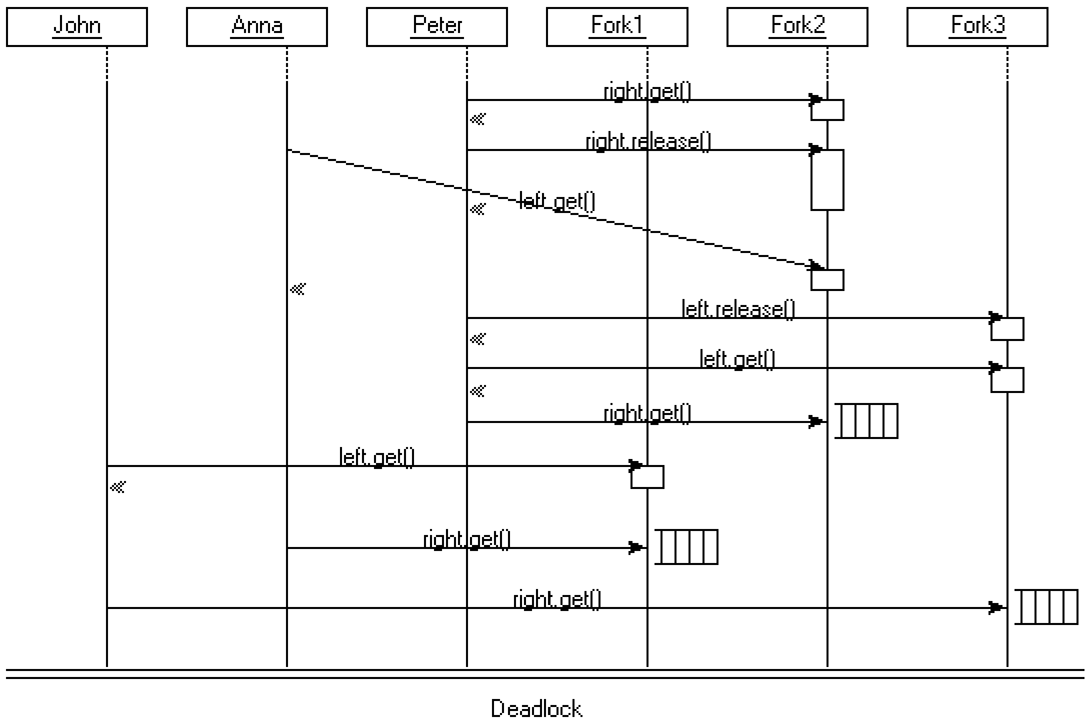


Figure 31: Result returned by vUML for the collaboration diagram of Figure 29 (Head of the sequence diagram is not represented.)

4.2.6 Conclusion

Though quite small in size, vUML already offers an impressively powerful bridge from UML models to PROMELA programs, and back from SPIN results to UML sequence charts. Unfortunately, because the tool is still in a prototype release, hardly any documentation is available on the particular syntax of the meta-model and on the source-code. Lilius and Paltor have published a study of a cell production case with vUML in [Lil3+99], which gives a good overview of the tool scope and performance, yet vUML does not seem to have been used so far in any industrial context.

A further important restriction of vUML is the impossibility of any dynamic instantiation of either links or objects [Lil2+99, p.18]. This means that the context and in particular the instances known to a given object are set in a static way by the collaboration diagram provided to vUML. As such they can't change dynamically during model execution. For example, in the previous dining philosopher example, a philosopher instance can't change the fork instance he is associated with after verification has started. As explained in the conclusion of [Lil2+99] they are very good reasons to forbid dynamic instantiations, in particular the increased complexity and the potential infiniteness of the resulting models. Though, this also considerably constrained the flexibility of Object Orientation, and it is certainly worth considering in which extend less restrictive approaches may be proposed. The Bandera tool, introduced in the last part of this section, is a representative of such a more adaptable method.

vUML is freely available on the Internet [*vUML00]. It is distributed under the GNU General Public License (GPL), which makes it an open source product. vUML requires a Python interpreter and a SPIN installation to work correctly. It can be called from a shell command line.

4.3 UMLAUT

4.3.1 Introduction

UMLAUT (*Unified Modeling Language All pUrposes Transformer*) is a general transformational framework for UML models. It is developed and maintained within the PAMPA project (*Modèles et Outils pour la Programmation des Architectures Parallèles Réparties*) under the direction of Jean-Marc Jézéquel at IRISA (*Institut de Recherche en Informatique et Systèmes Aléatoires*, Rennes, France) [HoJé+99]. UMLAUT is not directly dedicated to verification but rather to rewriting of UML models for various purposes (among others, the generation of verification models).

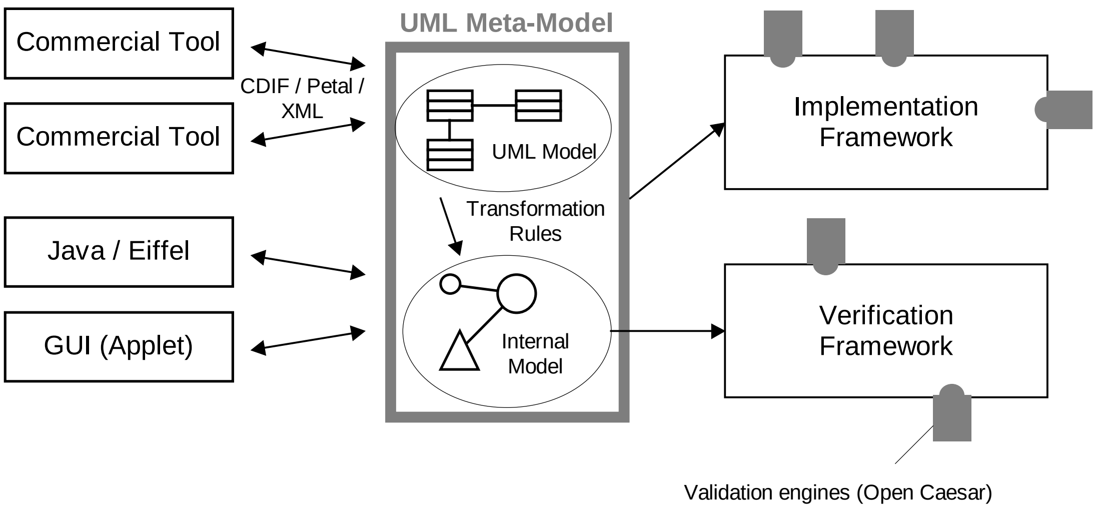


*Figure 32: UMLAUT Architecture (partial reproduction from [*Pamp00])*

4.3.2 Present and planned functionalities

UMLAUT is a wide scope project and is still in an early development phase. As a consequence, only beta versions of selected executables are publicly available. These releases already give a good idea of the envisioned final tool, but don't offer the full functionalities targeted by the project.

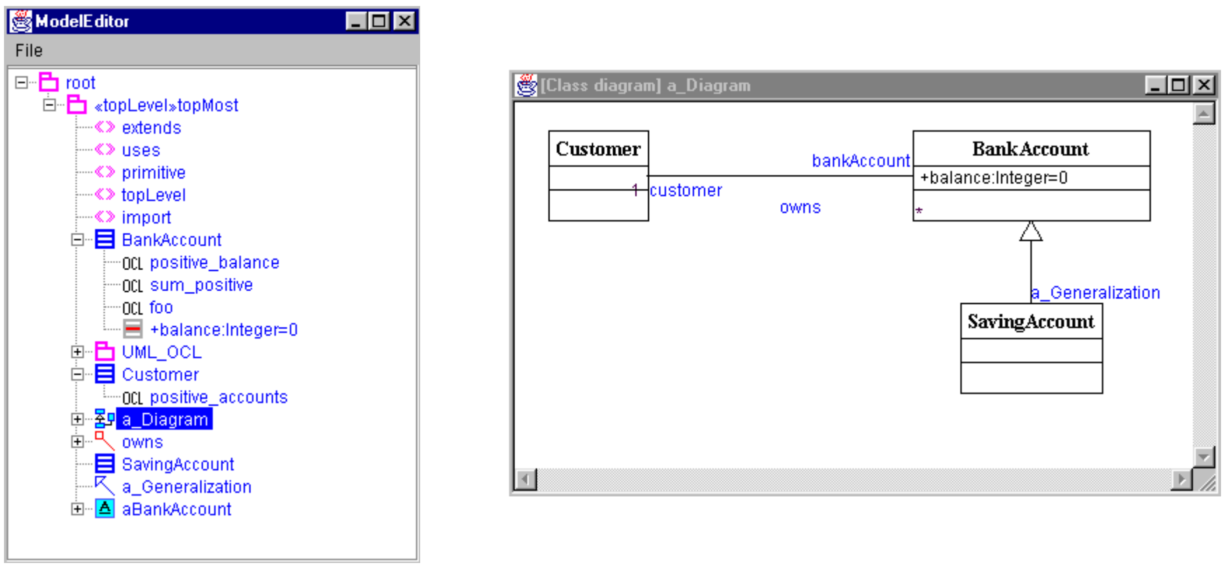


Figure 33: Model Browser and Class Diagram windows of UMLAUT

The versions available by the time of this study (Release Gauvain 1.5 and Release ASL 1.1) contains a clean implementation of the 1.1 metamodel along with some of the UML 1.3 extensions. It allows the graphic edition of UML models. Most diagrams are available, including State Charts diagrams. The Java Graphical User Interface suffers from some bugs, but that's quite understandable, considering the complexity of the tool. Code generation in Eiffel is available as well, along with OCL (Object Constraint Language) support for the specification of pre- and postconditions. Figure 33 shows two typical windows of UMLAUT displaying a sample model.

We could only test UMLAUT on NT. We had unfortunately some troubles with the Eiffel Code generation on that platform and could not test that feature.

In the meanwhile a new UMLAUT module for verification and simulation purposes has been released (UMLAUT/Simulator, Release Simulator 1.1), which offers a bridge to the CADP toolbox developed at INRIA [Gara+97] [*CADP00]. Unfortunately, we could not more precisely look at it because the tool was not yet available by the end of our study.

Though, an interesting question about that new module is the way a UML specification is converted to a finite state transition system. Actually, as discussed earlier about vUML, some essential features of most programming paradigms — like infinite recursion, or dynamic memory allocation —, and more specifically of object oriented notation — like dynamic class instantiation — introduce potentially unbounded behaviors, and thus ban any direct model checking. The question is which constrains, if any, UMLAUT sets on UML models to be able to convert them in finite discrete systems.

4.3.3 Availability and Conclusion

UMLAUT is planned to be distributed as freeware, once completed. At the time being only compiled executables may be downloaded from the IRISA web site [*Pamp00]. Those executables run with a java graphical environment and use a native shared library for core functionalities. Files may be read from either a native format, the CDIF format, or directly from Eiffel. Format conversion the other way round is also available, though this seems not to work very well under NT for the time being.

The UMLAUT system is available in three different versions: standard (Release Gauvain 1.5), with support for the Action Semantics Language [*OMG00] currently being defined at the Object Management Group (Release ASL 1.1), and as a simulator / translator to CADP (UMLAUT/Simulator, Release Simulator 1.1). Each version is available under Linux, Solaris and NT.

It is difficult to precisely evaluate the potentialities of UMLAUT at the present time, since most versions are still incomplete and are not completely stable. The root ideas behind the tool are very promising, and the generic approach that was chosen allows a great flexibility for a broad range of applications.

4.4 Bandera

4.4.1 Introduction

The last tool we look at within this overview is also the one we could the least experiment with. Bandera [Corb+00] [*Band00] is developed at the Laboratory for Specification, Analysis, and Transformation of Software at Kansas State University around Matt Dwyer, in collaboration with James Corbett of the University of Hawaii. Several publications related to Bandera have already been published, though the first public release under the GNU GPL license is not planned before September 2000.

The official web-site of the Bandera project presents the Bandera framework in the following way [*Band00]:

The Bandera project addresses one of the major obstacles in the path of practical finite-state of verification of software. Tools like, SMV and SPIN, accept as input a description of a finite-state transition system. Bridging the semantic gap between a non-finite-state software system expressed as source code and those tool input languages requires the application of sophisticated program analysis, abstraction and transformation techniques.*

The goal of the Bandera project is to integrate existing programming language processing techniques with newly developed techniques to provide automated support for the extraction of safe, compact, finite-state models that are suitable for verification from Java source code. While our ultimate goal is fully-automated model extraction for a broad class of software systems, our approach takes as a given that guidance from a software analyst may be required.

(source http://www.cis.ksu.edu/santos/bandera/ as on 23 August 2000)

4.4.2 Need for Reduction

The vUML tool presented before in this document introduces a totally defined mapping from OO models (expressed with UML) into the specific PROMELA language of the SPIN model checker. As mentioned before, this was only possible after Lilius and Paltor defined a clear formal semantics for UML, which was restricted enough to be expressed in the targeted PROMELA language. In the case of vUML, the source formalism (OO, and more precisely UML) itself was adapted to the target formalism (finite state transition models). As a consequence any model expressed with vUML is assured to be one-to-one convertible to a PROMELA program.

In the case of Bandera, the same approach can't be used, since the Java programming language can't be customized — and thus restricted when necessary — in the same way UML can. Because the expressiveness of Java is too powerful for direct model checking, each Java program that is to be verified by that mean must be boiled down to a finite state transition system in a first time. This is exactly the mission the Bandera Project set itself. Bandera provides a chain of more or less automated reduction mechanisms, which permit the conversion of a Java Source Code into the input languages of common model checkers. Figure 34 shows the architecture of the Bandera tool to reach that goal.

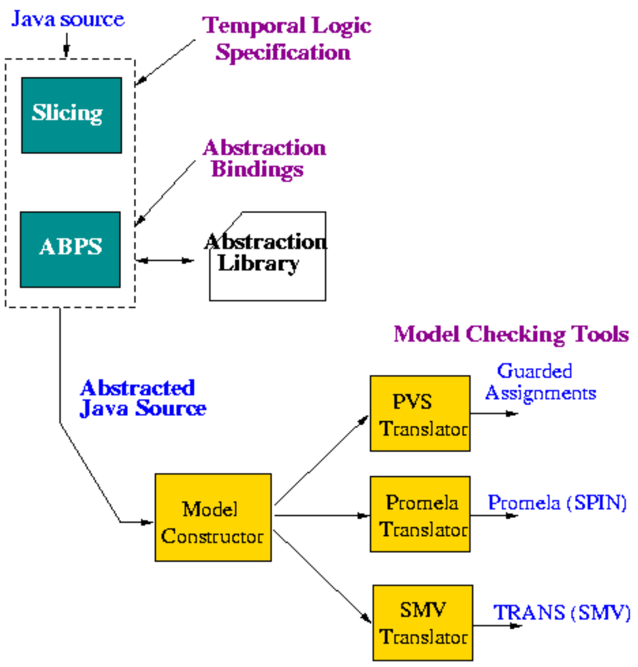


*Figure 34: Architecture of the Bandera Framework (reproduced from [*Band00])*

4.4.3 Slicing and Variable Abstraction Binding

The reductions provided by Bandera aim at producing an optimized abstracted model with respect to the property being checked. This goal is closely related to those of compilers and object code optimization, and because of that, the techniques proposed in Bandera are often inspired or related to those of compiler engineering.

The main idea is to get rid in a first time of any element (variables, threads) present in the source code that has no relevance to the property being checked. This first step is referred to as slicing. The result of slicing is then further reduced by applying data abstraction. The Bandera Data abstraction is similar to the SMV Data Abstraction mechanism. The ranges of values of some variables are interactively restricted to a smaller set of abstracted values. This process is referred to as abstraction binding, since each individual variable may be bound to a specific abstraction (provided abstraction compatibilities are respected). In order to allow abstraction reuse, Bandera provides a library of abstractions, which can be extended by the user. The undefined values and the indeterminism introduced by the data abstraction are automatically handled by Bandera.

Slicing and data abstraction may be sufficient in some case to get a finite tractable model. If this is not the case, model bounding is applied to limit the dynamic and potentially unbounded behavior of the model. (For instance by limiting the number of instances of a class.)

4.4.4 Conclusion

Similarly to UMLAUT, it's difficult to precisely evaluate Bandera, since no executable was released so far. Compared to vUML and UMLAUT, Bandera certainly goes a step further by trying to bridge the "semantic gap", as its authors call it, between programming languages and model checkers. We could also find another project focused on the verification of java program: JCAT (Java$^{(TM)}$ Concurrency Analysis Tool) is developed at the Computer Networks and Architectures Group at the Polytechnic of Turin [*JCAT00]. On a similar level the Bell Laboratories offer a tool called VERISOFT for systematic software testing using Model Checking [*Veri00]. We have not evaluated those tools, but they shows if needed that the direct connection between programming languages and verification tools has become an important research issue.

5 Conclusions

In this report, we have tried to give a broad overview of the model checking field, more particularly in connection with Object Oriented languages and notations. The main difficulty while trying to combine the better of the two worlds mainly arises from the irreducibility of Object Oriented Languages to any finite state transition structure. This semantic gap is actually not specific to Object Orientation in itself, and is typical of most programming languages. Object Orientation further complicates the matter by inherently allowing the dynamic instantiation of classes, and dynamic type substitution (polymorphism).

To work around that stumble stone, at least two ways have been investigated so far. The first one, represented by vUML and most probably by UMLAUT, consists in restricting the object oriented formalism up front. In particular vUML does not allow any run time instantiation. The second way, represented by Bandera, builds on compiler techniques to reduce an OO program into some finite discrete state transition system. This reduction may unavoidably result in some information loss, which must be controlled by the tool. In particular, results on the reduced system may not make any sense for the original java program because the reduction was too aggressive. This second approach is also naturally connected to Theorem Proving because of the inherent potential infiniteness of programs.

Both ways actually make sense in our opinion. The up-front restriction of formalisms implicitly introduces a process development philosophy in which certain design decisions are considered taboo a priori. It associates Model-Checking with a preventive approach to software development. One could argue on the benefit of Object Orientation if its most distinctive features must be disabled. This is definitely an issue deserving thorough discussion, which has more generally been at the core of most software engineering considerations. The second line of attack adopts a pragmatic philosophy. It tries to prove what can be, and accepts in advance the possible undecidability of the verification. This second approach allows more freedom in the design of software, and promises more gain in case of success. But success is not guarantied.

It is probable that any practical development process would make the best choice out of a combination of both approaches. We further see a great opportunity in the efforts to adapt formal model-checkers to popular OO Notations for the further democratizing of those techniques. With more practitioners given access to verification means, we can hope for a stronger industrial feedback on model-checking, and its benefits, as well as the ways it can best be applied in practice.

6 References

6.1 Bibliography

[Alur+94] R.ALUR, D.L. DILL, *A Theory of Timed Automata,*
Theoretical Computer Science, 126:183-235, 1994.

[Alur+93] R. ALUR, C. COURCOUBETIS, D.L. DILL, *Model-checking in dense real-time*.
Information and Computation 104(1):2-34, 1993.

[AscLu99] DAVID ASCHER, MARK LUTZ, *Learning Python*,
O'Reilly & Associates, March 1999, ISBN: 1-56592-464-9

[BCM92] J. R. BURCH, E. M. CLARKE, K. L. MCMILLAN, D. L. DILL, L. J. HWANG,
Symbolic Model Checking: 10^20 states and beyond,
Information and Computation, vol. 8, no. 2, June 1992, pp. 142-70.

[Bozg+98] M. BOZGA, C. DAWS, O. MALER, A. OLIVERO, S. TRIPAKIS, AND S. YOVINE,
KRONOS: a Model-Checking Tool for Real-Time Systems.
in Proceedings of the 10th Conference on Computer-Aided Verification, CAV'98, Vancouver, Canada, June 1998. LNCS 1427, Springer-Verlag

[Bryant86] RANDAL E.BRYANT, *Graph-Based Algorithms for Boolean Function Manipulation,*
IEEE Transaction on Computers, Vol. C-35, No. 8, August 1986

[Corb+00] JAMES CORBETT, MATTHEW DWYER, JOHN HATCLIFF, ET AL.
Bandera : Extracting Finite-state Models from Java Source Code, in Proceedings of the 22nd International Conference on Software Engineering, June, 2000.

[Gara+97] HUBERT GARAVEL ET AL., *CADP'97 - Status, Applications, and Perspectives*,
Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia), June 1997

[Hoare85] C.A.R. HOARE, *Communicating Sequential Processes*,
Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5

[HoJé+99] WAI-MING HO, JEAN-MARC JÉZÉQUEL, ALAIN LE GUENNEC, FRANÇOIS PENNANEAC'H.
UMLAUT: an extendible UML transformation framework.
In Proc. Automated Software Engineering, ASE'99, Florida, October 1999.

[Holz97] GERARD J. HOLZMANN, *The Model Checker* SPIN,
IEEE Transactions on Software Engineering, Special issue on Formal Methods in Software Practice, May 1997

[Lil1+99] JOHAN LILIUS, IVÁN PORRES PALTOR, *vUML: a Tool for Verifying UML Models*
TUCS Technical Report No. 272, May 1999, ISBN 952-12-0445-1, ISSN 1239-1891
available as PostScript at http://www.tucs.abo.fi/publications/techreports/TR272.html
(State 17th August 2000)

[Lil2+99] JOHAN LILIUS, IVÁN PORRES PALTOR, *The Semantics of UML State Machines*,
TUCS Technical Report No. 273, June 1999, ISBN 952-12-0446-X, ISSN 1239-1891,
available as PostScript at http://www.tucs.abo.fi/publications/techreports/TR273.html
(State as of 17th August 2000)

[Lil3+99] JOHAN LILIUS, IVÁN PORRES PALTOR,
The Production Cell: An Exercise in the Formal Verification of a UML Model, TUCS Technical Report No. 288, June 1999, ISBN 952-12-0480-X, ISSN 1239-1891
available as PostScript at http://www.tucs.abo.fi/publications/techreports/TR288.html
(State as of 18th August 2000)

[McMill97] KENNETH L. MCMILLAN, *A compositional rule for hardware design refinement*, Computer Aided Verification (CAV97), O. Grumberg Ed., Haifa Israel, June 1997, pp. 24-35.

[McMill92] KENNETH L. MCMILLAN, *Symbolic Model Checking, An approach to the state explosion problem*, Ph.D. Thesis, Carnegie Mellon University Technical Report CMU-CS-92-131, May 1992.

[Merz00] STEPHAN MERZ, *Model Checking*,
Tutorials of MOVEP'2k, Proceedings of the Summer School on MOdelling and VErification of Parallel processes. Nantes, 19-23 June 2000. Editors: F. Cassez, C. Jard, B. Rozoy, and M. Ryan. pp.52-70

[Rush00] JOHN RUSCHBY, *Theorem Proving for Verification*,
Tutorials of MOVEP'2k, Proceedings of the Summer School on MOdelling and VErification of Parallel processes. Nantes, 19-23 June 2000. Editors: F. Cassez, C. Jard, B. Rozoy, and M. Ryan. pp.71-84

[Yovine97] S.YOVINE, *KRONOS: A verification tool for real-time systems*, In Springer International Journal of Software Tools for Technology Transfer, Vol. 1, Nber. 1/2, October 1997.

[Yovine93] S.YOVINE, *Méthodes et Outils pour la Vérification Symbolique de Systèmes Temporisés*, Thèse pour l'obtention du titre de Docteur de l'Institut National Polytechnique de Grenoble, 19 mai 1993.

6.2 URLs

[*Band00] *Bandera : Software Model Construction for Finite-state Verification*, http://www.cis.ksu.edu/santos/bandera/, state as on 23 August 2000.

[*CADP00] *CADP Home page* , http://www.inrialpes.fr/vasy/cadp.html, state as on August 31rst 2000.

[*JCAT00] *JCAT- A Tool to Analyze Java Concurrent Programs*, http://www.dai-arc.polito.it/dai-arc/auto/tools/tool6.shtml, state as on August 31rst 2000.

[*Kron00] *Kronos Home Page*, http://www-verimag.imag.fr/TEMPORISE/kronos/, state as on August 31rst 2000.

[*McMill00] *Ken McMillan's Home Page at UCB*, http://www-cad.eecs.berkeley.edu/~kenmcmil/, state as on August 31rst 2000.

[*McMill99] KENNETH L. MCMILLAN, *The SMV Language*, Cadence Berkeley Labs, 2001 Addison St., Berkeley, CA 94704, USA, March 23 1999 http://www-cad.eecs.berkeley.edu/~kenmcmil/language.ps, state as on June 24th 2000

[*McM99b] KENNETH L. MCMILLAN, *Getting Started with SMV,* , Cadence Berkeley Labs, 2001 Addison St., Berkeley, CA 94704, USA, March 23 1999 http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial.ps, stand as on June 24th 2000

[*OMG00] *Action Semantics for UML RFP*, http://cgi.omg.org/techprocess/meetings/schedule/Action_Semantics_for_UML_RFP.html, last updated: Monday, July 10th 2000

[*Pamp00] *UMLAUT Home Page*, http://www.irisa.fr/pampa/UMLAUT, state as on August 31rst 2000.

[*Pyth00] *Python Language Website*, http://www.python.org/, state as on August 31rst 2000.

[*Spin00] *Spin - Formal Verification*, http://netlib.bell-labs.com/netlib/spin/whatispin.html as last modified on 5th April 2000.

[*UML1.3] UML REVISION TASK FORCE, *OMG UML v. 1.3*, Object Management Group, http://www.omg.org/cgi-bin/doc?ad/99-06-08.pdf, 8 June 1999.

[*Upp00] UPPAAL2k http://www.docs.uu.se/docs/rtmv/uppaal/, state as on June 28th 2000.

[*Veri00] *VeriSoft Home-Page*, http://www1.bell-labs.com/project/verisoft/, state as on August 31rst 2000.

[*vUML00] *The vUML Page*, http://www.abo.fi/~iporres/vUML/, state as on August 16th 2000.