Modeling complex systems with VeriJ

Yan Zhang
Université Pierre & Marie Curie,
CNRS-UMR 7606 (LIP6/MoVe),
4 place Jussieu, F-75005 Paris, France
Yan.Zhang@lip6.fr

Béatrice Bérard
Université Pierre & Marie Curie,
CNRS-UMR 7606 (LIP6/MoVe),
4 place Jussieu, F-75005 Paris, France
Beatrice.Berard@lip6.fr

Lom Messan Hillah
CNRS-UMR 7606 (LIP6/MoVe) and
Université Paris Ouest Nanterre La Défense
200, avenue de la République, F-92001 Nanterre Cedex, France
Lom-Messan.Hillah@lip6.fr

Fabrice Kordon
Université Pierre & Marie Curie,
CNRS-UMR 7606 (LIP6/MoVe),
4 place Jussieu, F-75005 Paris, France
Fabrice.Kordon@lip6.fr

Yann Thierry-Mieg
Université Pierre & Marie Curie,
CNRS-UMR 7606 (LIP6/MoVe),
4 place Jussieu, F-75005 Paris, France
Yann.Thierry-Mieg@lip6.fr

This paper presents VeriJ, a language designed for modeling complex supervisory control problems. VeriJ is based on a subset of the Java language with some supervisory control specific constructs added; this allows to use industrial strength integrated development environments such as Eclipse to build VeriJ models and to directly use a Java debugger to execute (simulate) these models. With the aim to perform controller synthesis in a further step, VeriJ models are translated into hierarchical finite state machines (HFSM) representing the control flow graph, using modern model transformation techniques and tools. The semantics of these HFSM is then given as a pushdown system, leading to a concise and expressive representation of the underlying discrete event system. We illustrate our modeling and transformation approach with a VeriJ model of the Nim game, for which finding a winning strategy for a player can be seen as a control problem.

Keywords: VeriJ, Java, model transformation, verification, controller synthesis

1. INTRODUCTION

Context. Supervisory control[1] of discrete event systems is a formal approach allowing to automatically compute a controller given some control objective. Given a discrete model $\mathcal{M}$ of a system and an objective expressed as a formula φ, the control problem asks if there exists a controller $\mathcal{C}$ such that $\mathcal{M}$ controlled by $\mathcal{C}$ satisfies φ. This problem can also be viewed as a game where the controller is looking for a winning stategy against all possible actions of the environment.

While algorithms solving the control problem are well known (Ramadge and Wonham 1987), two obstacles limit practical application of these techniques in industry: like most state-space exploration techniques the algorithms scale up with difficulty to large and complex systems, and from an engineer's point of view the investment needed to learn to manipulate formal models such as automata is often considered too costly.

Contribution. We propose to model complex discrete event systems using VeriJ (Zhang 2010), a language based on a subset of Java. Complex systems may involve a large number of components and handle for instance lists with dynamic size. They would include automated transport systems like the one partially studied in (Bérard et al. 2008). Concurrency is an important feature of complex systems but it is not yet implemented and is left out in this paper. In addition to Java instructions, VeriJ includes a small set of control specific elements allowing to specify which actions are controllable, and the control objective. Since this language is based on Java, most engineers already feel very comfortable expressing their system's semantics as programs. This also allows to directly benefit from mature and powerful industrial strength development

[1]This work has been partially supported by a Ph. D. grant from the Chinese Scholarship Council.

environments such as Eclipse (syntax checks, code completion, etc.), including the facilities to interactively run a VeriJ model thanks to the standard Java debugger.

Discussion and related work. To perform formal analysis of VeriJ specifications, we need to build a transition system from the java-based source code. Two main alternative solutions are possible for this step: direct translation from source code to a formal model, or using the Java compiler to obtain bytecode, then analyze at bytecode level. We now compare existing techniques used in related work and present our proposal.

Direct translation to some variant of Control Flow Graph (CFG) allows to preserve a high level of abstraction in the resulting system model. However, it may be difficult in general to capture all syntactic elements from the language, and care must be taken to avoid deviation from a compiler's interpretation of the source code. This approach was used for instance in the Bandera project (Corbett et al. 2000) where the translation target was a so called Bandera Intermediate Representation akin to a finite state machine. It was also the case in early versions of the Java Path Finder (JPF) (Havelund 1999), a software model-checker with Promela as target: Promela is the input language of Spin model-checker (Holzmann 1997), again based on communicating finite automata.

The other option consists in using a standard compiler to derive the semantics of the source code, and handling the verification by working at the bytecode (*e.g.* for Java) or assembly language (*e.g.* for C) level. This approach solves issues related to software artifacts, such as external libraries for which no source code is available, hence is the preferred option for full-fledged software model-checkers. It also allows to consider less cases when implementing the verification tool as the variety of opcodes is rather limited. However, it forces to work at a very low level of abstraction, on much larger models in raw number of instructions, or even to resort to executing the code to derive its interpretation. This is the choice taken in recent versions of JPF (Brat et al. 2000; Gvero et al. 2008) which rely on a dedicated backtrackable Java Virtual Machine (JVM) that provides non-deterministic choices and control over thread scheduling.

In this work, we choose to directly translate the VeriJ input into a variant of control flow graphs called Hierarchical Finite State Machines (HFSM), that preserve the structure of the source code and a high level of abstraction. Parsing of the input is partially handled by existing Java analysis tools: MoDisco[2] is able to raise the source code to a model instance of a standard Java metamodel. Subsequent transformation to HFSM relies on model to model (M2M) transformation techniques using the Atlas Transformation Language (ATL[3]), a state-of-the-art model transformation plugin within Eclipse. Since we are targeting supervisory control rather than full software model-checking, we need an efficient expression of the system's transition relation. For this reason, JVM-based solutions (like in JPF) are not appropriate in our case.

Once a control flow graph has been obtained, there remains the question of how to provide its semantics. A straightforward approach in simple cases consists in inlining all calls, resulting in a single finite state machine (FSM) for the whole input program. FSMs are the natural input language for many model-checkers, which makes this solution attractive. However, a plain FSM may become very large due to duplication of behaviors. Moreover, it is inapplicable when recursion is involved, since the inlining would produce infinite structures. Another popular approach is to use pushdown semantics to interpret the CFG: using PushDown Systems (PDS) produces a compact and accurate representation of procedure calls thanks to the use of a stack in the system states. This is the choice taken in jMoped (Suwimonteerabuth et al. 2007), Magic (Chaki et al. 2006) or MOPS (Chen and Wagner 2002) for instance, and in our own approach as well.

Outline. The overall scheme of our approach is depicted in Figure 1. The transformation is split into two steps: preprocessing and compilation, both using model transformation techniques involving metamodels.

Instead of a really complex system, we use in this paper as a running example a simple two-player game, Nim, where the control problem consists in finding a winning strategy for one of the players.

The paper is structured as follows. Section 2 presents VeriJ and its metamodel. Section 3 is devoted to VeriJ compilation into Hierarchical Finite State Machines then Section 4 shows the pushdown semantics.

2. VERIJ

VeriJ is meant to bridge the gap between a programming language and the input of a controller synthesis tool.

[2] `www.eclipse.org/MoDisco/`
[3] `www.eclipse.org/atl`

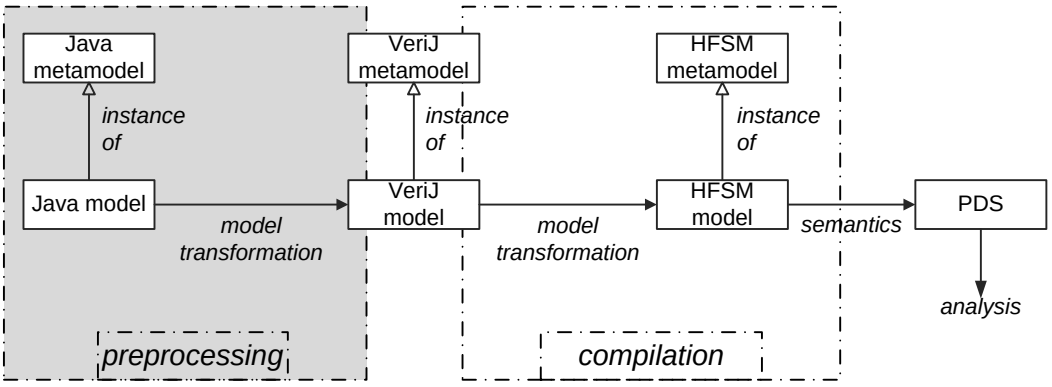


Figure 1: *From source code to formal model.*

2.1. VeriJ definition

Designed as a Domain Specific Language (DSL) for verification and control synthesis, VeriJ consists of a subset of the Java language, including elements such as: basic data types, arithmetic operators, assignments, decision and control statements, construction of classes with instantiation, with the addition of specific classes described below. VeriJ does not support features such as cast, exceptions, visibility, inheritance, libraries or native code (see Figure 2(a)).

We now present VeriJ specific features, shown in Figure 2(b).

- Due to the complexity of dealing with low-level Java collections such as sets, lists and so on, we create the `VeriJList` type instead, to handle basic collections with a small set of operations. For example, in Java, even the basic collection ArrayList involves hundreds of lines of complex source code. The `VeriJList` provides a high level of abstraction for the collections.
- Additionally, VeriJ introduces concepts useful for the verification process, such as *random* and *non-deterministic choice* (*NDChoice*). In VeriJ, `random` is a random integer generator built on the standard `Math.random()`, which randomly produces an integer between a `min` and a `max`. The `NDChoice` method is a random boolean generator. Both `random` and `NDChoice` are used to specify free choice semantics of a system and will be used to build the transition relation of the target model. The `playerID` parameter given in these two methods is needed to identify by whom each choice is made. In other words, each move is labeled by its player, which will be an essential part in the procedures of verification and controller synthesis.

These concepts are also implemented as Java classes, hence allowing us to run and debug VeriJ models using standard Java tools. In our implementation of the random methods, user input, trace record and replay or standard simulation are possible.

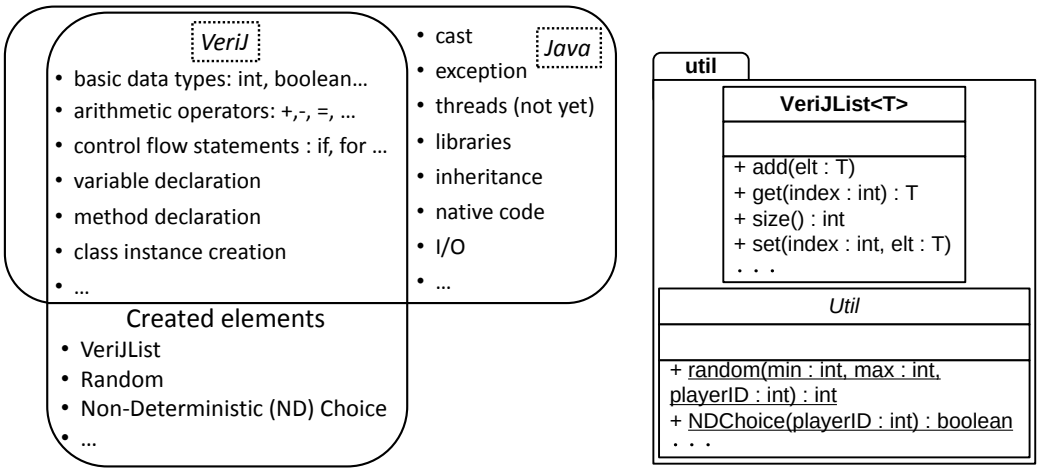
2.2. Example of a VeriJ model: the Nim game

We now present the Nim game, used as a running example throughout the paper. In this case, we would like to solve a control problem: finding a winning strategy for one of the player. In the final transition system, each action must be assigned to a player and a set of failure states have to be defined. Then a standard algorithm involving a backward fixpoint will be applied as in (Zhang et al. 2010).

Given a set of matches arranged in several rows, with $2i - 1$ matches in the i^{th} row, the Nim game consists of two players alternately picking a random number of matches from a randomly selected row of matches. The player who takes the last match loses. This game was completely solved in 1901 by Charles Bouton (Bouton 1901). We use here the variant presented in (Ziller 2002). Such a game can be seen as an instance of a controller synthesis problem: the game is modeled as a transition system where each player takes a turn and the goal is to determine if there exists a winning strategy for one of the two players. In other words, one of the players, the *controller*, tries to find a winning strategy against all the possible moves of the other player, who represents the *environment*.

The Nim game source code being large, we only present a subset in Figure 3, to show how modeling and simulation are done in VeriJ.

- The class `Board` only uses a variable `Matches`, whose type is `VeriJList`. Its constructor (lines 36 to 42) constructs the set of matches in successive rows by calling operation `add` (from



(a) VeriJ versus Java. (b) VeriJ additional elements

Figure 2: *Illustration of VeriJ definition.*

```
1 public class TestNim {
2  public static void
3      main(String[] args) {
4   Nim nim;
5   nim = new Nim();
6   boolean gameover;
7   gameover = nim.gameover();
8   while (!gameover) {
9    nim.Transition();
10   gameover = nim.gameover();
11  } }
12}
13
14public class Nim {
15  private Board board;
16  private int playerID;
17
18  public Nim(){
19   board = new Board(); }
20  public void Transition() {
21   boolean gameover;
22    gameover = gameover();
23   if (!gameover) {
24    playEnvironment();
25   }
26   if (!gameover) {
27    playController();
28   } else
29    isBad();
30 } ...
31}
32
33public class Board {
34  private List<Integer> Matches;
35  private boolean isMatchEmpty;
36 public Board() {
37  Matches = new VeriJList<Integer>();
38  for (int i = 0;
39          i < Constants.NBRow; i=i+1) {
40   int num;
41   num = 2 * i + 1;
42   Matches.add(num);}}
43 private int chooseNBtake(int index,
44     int playerID) {
45  boolean empty;
46          empty = matchEmpty();
47  if(empty)
48   return 0;
49  else{
50   int matchesNumber;
51   matchesNumber = Matches.get(index);
52   int maxTake;
53   maxTake = Math.min(matchesNumber,
54              Constants.MaxNBtaken);
55   int random;
56   random = Util.random(1,maxTake, playerID);
57   return random; }}
58 private int chooseRow(int playerID) {
59  int NonemptyRow = -1;
60  int size;
61   size = Matches.size();
62  for (int NBMatchRow = 0; NBMatchRow < size;
63          NBMatchRow=NBMatchRow+1) {
64   int getNBMatchRow;
65    getNBMatchRow = Matches.get(NBMatchRow);
66   if (getNBMatchRow != 0) {
67    NonemptyRow = NBMatchRow;
68    boolean choice;
69     choice = Util.NDChoice(playerID);
70    if (choice)
71     return NBMatchRow;
72  }}   ... } ...}
```

Figure 3: *VeriJ description of the Nim game.*

the `VeriJList` type), with argument $(2 * \mathtt{i} + 1)$ within a loop.

- A call to `random` (line 56) is used in method `chooseNBtake` to randomly generate the number of matches to take in one move. Similarly, `NDChoice` (line 69) is used in method `chooseRow` to randomly choose the row in which matches will be taken in that move. Both `random` and `NDChoice` carry the label of each player by `playerID`.

Figure 4 shows the class diagram of the Nim game from VeriJ source code. It contains four classes: TestNim, Nim, Board and Constants. Classes Nim and Board constitute the core part of the model. This model was produced from the VeriJ code using a standard UML tool.

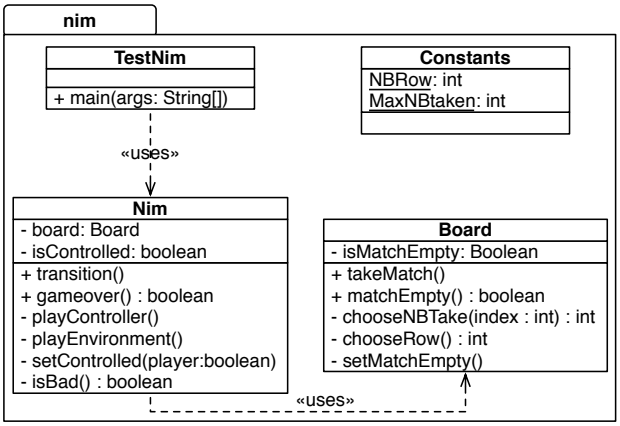


Figure 4: *Class diagram of the Nim game.*

2.3. VeriJ compilation

To apply formal analysis to VeriJ models, VeriJ source code is compiled into a Hierarchical Finite State machine (HFSM), *i.e.* a set of finite state machines. Recall that the transformation (Figure 1) includes preprocessing and compilation, using model transformation techniques involving metamodels.

Since VeriJ takes a subset of Java as described in Figure 2(a), its metamodel is primarily derived from the metamodel of Java, by removing 46 metaclasses (from the 126 metaclasses of Java) and adding the new elements. Despite the classes removed from Java metamodel, VeriJ metamodel is still too large to be displayed in this paper. To get an in-depth description and complete details of the Java metamodel, we refer the reader to `http://wiki.eclipse.org/MoDisco/Components/Java/Documentation/0.9`.

In the preprocessing step, we extract the Java model of the application, conforming to Java metamodel, from either the Java source code or the VeriJ source code, thanks to MoDisco. MoDisco is a model-driven framework providing tools to support software modernization. Among these tools, discoverers automatically create models of existing systems. The VeriJ model, conforming to VeriJ metamodel, built out of the extracted Java model, is obtained by pruning unnecessary information from the Java model and building the VeriJ specific elements. Given the Java metamodel and the VeriJ metamodel, this step is carried out by a set of rules `Java2VeriJ.atl`, using Atlas Transformation Language (ATL) framework. A part of the transformation code is shown in Figure 6. Rule `VeriJRandom` (line 9 to 18) shows how to select and transform a `MethodInvocation` element in Java to a `random` element in VeriJ.

```
1-- @path MMJava=/Java2VeriJ/java.ecore
2-- @path MMVeriJ=/Java2VeriJ/veriJ_02.ecore
3module Java2VeriJ;
4create OUT : MMVeriJ from IN : MMJava;
5
6helper context MMJava!MethodInvocation def :
7isVeriJMethod() : Boolean = self.method.proxy;
8
9rule VeriJRandom{
10 from s : MMJava!MethodInvocation(
11     s.isVeriJMethod() and s.method.name.
12     startsWith('random'))
13 to t : MMVeriJ!random(
14  min <- s.arguments.at(1),
15  max <- s.arguments.at(2),
16  playerID <- s.arguments.at(3)
17 )
18}
```

Figure 6: *ATL Rule for `VeriJRandom`.*

3. HIERARCHICAL FINITE STATE MACHINE

The next step of our approach is to compile VeriJ models into discrete event systems (Figure 1). The control flow of a VeriJ model can be syntactically described by a Hierarchical Finite State Machine (HFSM), which is a finite set of finite automata, linked together according to program instructions. This model and its semantics in terms of a pushdown system is intended to be the input of the verification tool. We first describe HFSM with the corresponding model transformation and give the pushdown semantics in Section 4.

3.1. Definition

A finite state machine (FSM) over a finite alphabet Σ is a tuple $F = (S, \delta, s_0, s_f)$ where S is a finite nonempty set of *states*, δ is a partial mapping from $S \times \Sigma$ to S, $s_0 \in S$ is the *initial state* and s_f is a unique *final state*.

Definition 1 *A Hierarchical Finite State Machine (HFSM) is a finite set $\mathcal{F}$ of finite state machines, with*

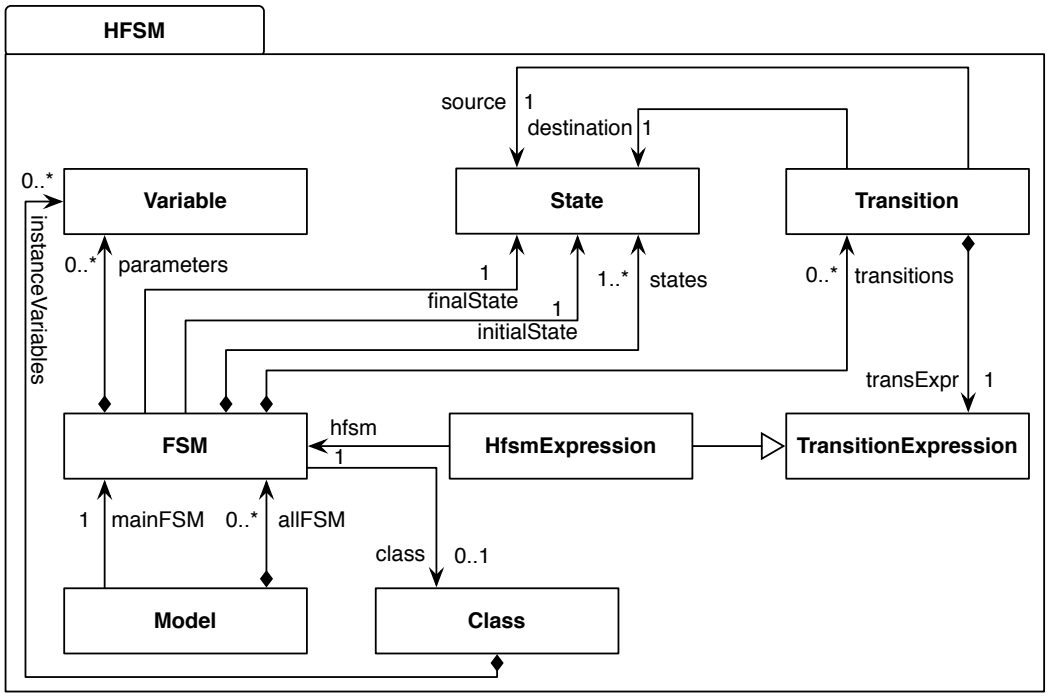


Figure 5: *Core part of the metamodel of HFSM.*

an initial FSM $F_0 \in \mathcal{F}$ and alphabet $\Sigma \cup \{r_F, F \in \mathcal{F}\}$, where Σ is a finite alphabet and each r_F is a symbol not in Σ.

The initial FSM F_0, which represents the $main$ method, is the entry of $\mathcal{F}$. States s_0 and s_f of F_0 are respectively the initial and final states for the HFSM $\mathcal{F}$. Each finite state machine $F \in \mathcal{F}$ corresponds to a function called by the program executed from the $main$ method. A transition within F corresponds to an instruction of the function. When the instruction is a method invocation, the label of the transition is a reference to another FSM, and so on. Thus letter r_F denotes the *reference* to FSM F, while basic instructions are elements of Σ.

Figure 5 shows the core part of the HFSM metamodel, the actual complete metamodel being much larger. A model is composed of a set of FSMs. Each FSM is composed of states and transitions, together with its own local variables, and a class name (corresponding to the class to which it belongs). A **Transition** has a source state, a destination state and a **TransitionExpression**. The **TransitionExpression** is an expression denoting a simple statement (*e.g.* variable declaration, assignment, etc.) or an **HfsmExpression** (method invocation) which refers to an FSM, hence bringing in the hierarchy.

In HFSM metamodel, both the types of parameters of a method declaration and the fields of a class belong to the **Variable** metaclass. Figure 7 illustrates details of this element, which contains three subclasses: IntVar, BoolVar and ObjectVar which represent the three types Integer, Boolean and References to objects. A boolean tag is associated with variables to indicate if the variable is global (tag is true) or local. This feature is used in the semantics of the HFSM.

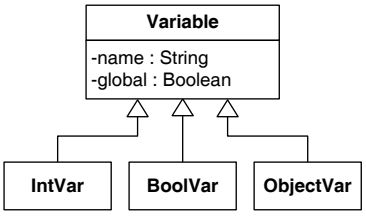


Figure 7: *Variable in HFSM metamodel.*

3.2. From VeriJ to HFSM

As mentioned above, the compilation of VeriJ (Figure 1) consists in the model transformation from a system specified in VeriJ to its HFSM model. Using VeriJ metamodel and HFSM metamodel, we code a set of ATL rules `VeriJ2Hfsm.atl` to create the states and transitions for each FSM in this HFSM model. To visualize the obtained HFSM model described in the form of XML Metadata Interchange, we create the corresponding `.dot` file through project *FSM2Dot*[4] and then generate the hierarchical FSM diagrams as shown in Figure 8 using Graphviz[5].

[4] `https://srcdev.lip6.fr/trac/research/yzhang/browser/FSM2Dot`
[5] `http://www.graphviz.org/`

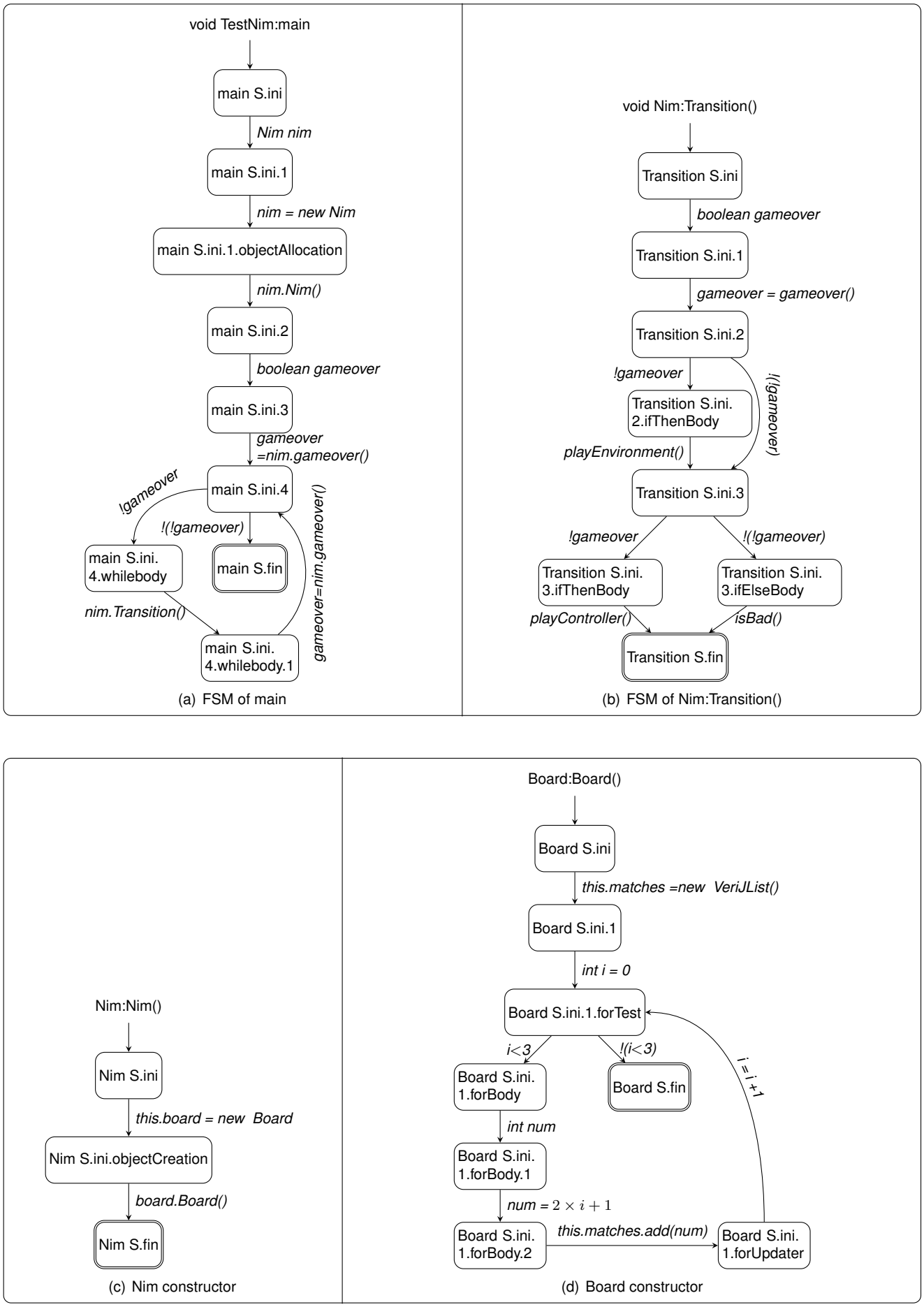


Figure 8: *HFSMs of Nim game obtained by application of* `VeriJ2HFSM.atl`.

The transformation associates with each method of the program an FSM with the same name, with initial and final states, which are called respectively by the FSM's name followed by "S.ini" and "S.fin".

Each transition is obtained from a statement of the VeriJ model. It takes the name of the statement as the label "transExpr" of **TransitionExpression** type. In particular, since a block statement in VeriJ is a special subclass of Statement which contains statements, each block is transformed into a list of transitions in an FSM, for example, the body of a method declaration, the body of the control statements such as If Statement (without else statement from *Transition S.ini.2* to *Transition S.ini.3* and with else statement from *Transition S.ini.3* to *Transition S.fin*), While Statement (from *main S.ini.4* to *main S.fin*) and For Statement (from *Board S.ini.1* to *Board S.fin*).

The naming of the states inside a FSM is defined in the following way: Given the source and destination states of the transition obtained from a Block statement, the states in the list of transitions are named by adding the ordered number or strings that indicate the structure of a control statement. For example, in Figure 8(a), *main S.ini.1* denotes the source state of the first transition, *main S.ini.4.whilebody.1* represents the source state of the Block statement, the while body.

3.3. Example of the Nim game

The result of the transformation is now illustrated on the Nim game example from Section 2.2. A part of the associated set of FSMs is depicted in Figure 8. We give details about the transformation on this example, refering to pieces of code from Figure 3.

Figure 8(a) present the main FSM in the class `TestNim`. Figure 8(c) and Figure 8(b) are from the class `Nim` presenting constructor and the function *Transition* respectively. Figure 8(d) gives the constructor of class `Board`. We now describe the hierarchical structure of these FSMs.

In Figure 8(a), the invocations of constructor `Nim()` from *S.ini.1* to *S.ini.2*, `nim.gameover()` from *S.ini.3* to *S.ini.4* and `nim.Transition()` from *S.ini.4.whilbody* to *S.ini.4.whilbody.1* compose the first level of the hierarchy. For instance, the method invocation of `Transition` reference the FSM in Figure 8(b), while the constructor invocation transition expression references the FSM in Figure 8(c). In addition, the invocation of `Board()` referencing to the FSM in Figure 8(d) makes the second level of the hierarchy.

The two steps above thus set up a consistent chained model transformation framework which makes maintenance and refinement easy. ATL also provides traceability mechanisms of the transformations in the form of TraceAdder (Jouault 2005). Another Eclipse plug-in from Atlas group, Atlas Model Weaver (AMW[6]), also offers a means to generate an ATL execution trace in a weaving model. Once the HSFM model is generated from the VeriJ model, we can build the corresponding pushdown system.

4. FROM HFSM TO PUSHDOWN SYSTEMS

It is well known that FSMs are not expressive enough to describe program semantics. For example, method calls and returns need to be correctly matched, including recursion schemes. Local variables in different procedure calls need to be distinguished. Pushdown systems (PDS), introduced in (Oettinger 1961; Chomsky 1962), are a natural choice for modeling method calls and interprocedural program behaviours, by adding a (possibly unbounded) stack to a finite set of control states. As mentioned in the introduction, pushdown systems are used in several recent verification tools (*e.g.* jMoped for Java and MOPS or Magic for C) to describe programs when recursion is involved. In fact, pushdown systems give a formal semantics to programs.

4.1. Definitions

The basic definition of pushdown automata is the following.

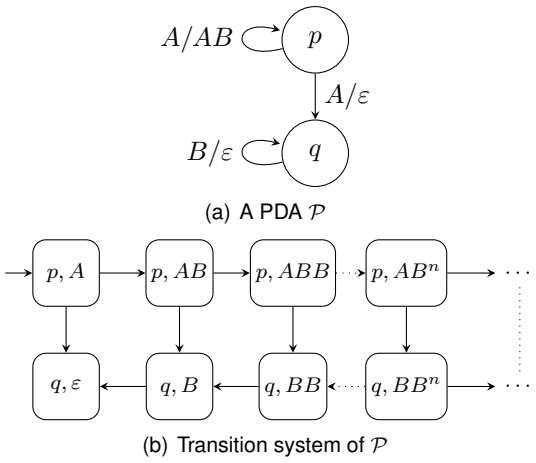
Definition 2 *A Pushdown Automaton (PDA) is a tuple* $\mathcal{P} = (P, \Gamma, \Delta, c_0)$*, where* P *is the set of states or control locations,* Γ *is the alphabet of stack symbols,* Δ *is the set of transition rules, a partial mapping from* $P \times \Gamma$ *to* $P \times \Gamma^*$*, and* $c_0 \in P \times \Gamma$ *is the initial configuration. A transition rule* $\delta \in \Delta$ *is written as* $(p, z) \hookrightarrow (p', \alpha)$ *with* $p, p' \in P, z \in \Gamma$ *and* $\alpha \in \Gamma^*$*.*

The semantics of $\mathcal{P}$ is given as a transition system with $P \times \Gamma^*$ the set of configurations. For a rule $\delta \in \Delta$ and a non empty $\gamma = z\beta \in \Gamma^*$ with $z \in \Gamma$ and $\beta \in \Gamma^*$, there is a transition $(p, \gamma) \xrightarrow{\delta} (p', \alpha\beta)$. In other words, z is the topmost stack symbol, each transition pops z and pushes the word α.

Figure 9 shows an example of a pushdown automaton with its semantics. In Figure 9(a), $P = \{p, q\}$, $\Gamma = \{A, B\}$, Δ contains the three rules $(p, A) \hookrightarrow (p, AB)$, $(p, A) \hookrightarrow (q, \varepsilon)$ and $(q, B) \hookrightarrow (q, \varepsilon)$, and the initial configuration is $c_0 = (p, A)$. The

[6] `http://www.eclipse.org/gmt/amw`

PDA has an infinite set of configurations depicted in Figure 9(b).



(a) A PDA $\mathcal{P}$

(b) Transition system of $\mathcal{P}$

Figure 9: *A PDA example.*

4.2. Extended PDS

In order to represent object references, we slightly extend the definition of pushdown automata to include an explicit representation of the heap.

Let X be a set of variables, including variable $this$. For each $x \in X$, let D_x be the (finite) range of x. The set D_x can either be the set of values of a primitive type (restricted to `int` and `boolean` here) or the set of references $Ref = \{\$0, \$1, \$2, \ldots\}$ containing heap addresses. In particular, $D_{this} = Ref$. Default values are $\perp$ for all ranges, for instance 0 for `int`, $false$ for `boolean` or $\$0$ for Ref.

A valuation is a partial mapping $v : X \to D$ where $D = \bigcup_{x \in X} D_x$ such that for each $x \in X$, $v(x) \in D_x$. For a valuation v, we define $Dom(v) = \{x \in X \mid v(x) \text{ is defined}\}$ and we denote by V the set of all valuations, with $\emptyset$ for the valuation such that $Dom(v) = \emptyset$. For $y \in X$ and $d \in D_y$, define v' by: $Dom(v') = Dom(v) \cup \{y\}$, $v'(y) = d$ and $v'(x) = v(x)$ for $x \neq y$. If $y \notin Dom(v)$, we write $v' = v \cup [y \mapsto d]$ and if $y \in Dom(v)$, we write $v' = v[y \mapsto d]$. Along the same line, for a subset $\{x_1, \ldots, x_n\}$ of X and values $d_1, \ldots, d_n$, we denote by $[x_1 \mapsto d_1, \ldots, x_n \mapsto d_n]$ the valuation v such that $Dom(v) = \{x_1, \ldots, x_n\}$ and $v(x_i) = d_i$ for $i = 1, \ldots, n$.

Given a HFSM $\mathcal{F}$ with initial FSM F_0, we write $\hat{\Sigma} = \Sigma \cup \{r_F, F \in \mathcal{F}\}$ and $F = (S_F, \hat{\Sigma}, \delta_F, s_{0,F}, s_{f,F})$ for each $F \in \mathcal{F}$. We set $S = \bigcup_{F \in \mathcal{F}} S_F$, the stack alphabet is $\Gamma = S \times V$ (recall that V is the set of all valuations) and we define the set of configurations by: $Q = (C \times V)^* \times V \times \Gamma^*$ where C is the set of class names from the VeriJ model (with $\emptyset$ for the empty class name). A configuration $q = (h, g, \gamma) \in Q$, consists of:

- $h \in (C \times V)^*$ the *heap state*. Hence, an empty heap is described by ε (also represented in the figure by $\$0 : \perp$) and the letters are of the form $(c, w) \in C \times V$, where c is a class name and $w \in V$ is a valuation for the attributes of an object in this class.
  A non empty heap is described either as $h = (c_1, w_1) \ldots (c_n, w_n)$ for some $n \geq 1$ and adding an object to h can be written as $h.(c, w)$ for some new $(c, w) \in C \times V$.

- $g \in V$ the *global variable state* is the valuation of static variables and the temporary variables for return statements; For a variable in g, the boolean tag *global* in **Variable** has value true.

- $\gamma \in \Gamma^*$ the *stack state* where each element $(s, v) \in \Gamma$ is composed of an FSM location and a variable valuation.

We now explain how the HFSM is interpreted in terms of PDS actions. The initial configuration of the PDS is $q_0 = (\varepsilon, \emptyset, (main_0, \emptyset)$, where $main_0$ denotes the initial state of F_0. A transition from configuration $q = (h, g, \gamma)$ to configuration $q' = (h', g', \gamma')$ is written $q \xrightarrow{t} q'$, for some transition $t : s \xrightarrow{\delta} s'$ from FSM F with $\delta \in \hat{\Sigma}$. The stack state evolves from $\gamma = z\beta$ to $\gamma' = \alpha\beta$, where $\alpha \in \Gamma^*$ is a word of length $|\alpha| \leq 2$. In this context, it is sufficient to consider rules with maximal length 2, which corresponds to a method invocation: stacking the initial state of the method called and the return address after popping the topmost stack symbol.

We finally give several examples of rules from configuration $q = (h, g, \gamma)$, assuming the size of the heap is $|h| = n$ and $\gamma = z\beta$, with $z = (s, v)$ the topmost stack symbol.

(i) Non-static variable declaration **of the form**
$\delta : type\ x = initializer;$
This statement adds to v a valuation for x, assigning the result d of the evaluation of $initializer$, which changes the stack state. The successor state is $q' = (h, g, \gamma')$ with $\gamma' = \alpha\beta$ and $\alpha = (s', v \cup [x \mapsto d])$.

(ii) Instantiate a class **of the form** $\delta : new\ Class;$
For a variable x declared as a reference variable, this operator allocates memory for the new object in the heap and returns a reference to that memory cell. The successor state is $q' = (h', g, \gamma')$ with $h' = h.(c, w)$ where c is the class name $Class$ and w assigns default values to all fields. Moreover, $\gamma' = \alpha\beta$ with $\alpha = (s', v \cup [x \mapsto \$(n+1)])$. Instantaneously, another rule (Method invocation) is applied for the constructor call.

***(iii) Method invocation* of the form**
$\delta : ob.m(arg_1, arg_2...)$;
Let M be the FSM associated with method $m()$ with m_0 its initial state, m_f its final state and $x_1, x_2...$ the parameters. The successor state is $q' = (h, g, \gamma')$ with $\gamma' = \alpha\beta$ and $\alpha = ((m_0, v_0)(s', v))$, where v_0 is the valuation defined by $v_0(x_i) = arg_i$ for $i = 1, 2, \ldots$ and $v_0(this) = \$j$ if the reference of object ob is $\$j$. Note that reaching the final state m_f of M (with a return statement) will pop the topmost stack symbol, hence returning the control to s', the successor state within the FSM that emitted the call.

***(iv) Assignment* of the form** $\delta : x = expression$;
Let d be the value resulting from the expression evaluation. When x is a field stored in the heap cell (c, w), the successor is $q' = (h', g, \gamma')$ where h' changes (c, w) to $(c, w[x \mapsto d])$, $\gamma' = \alpha\beta$ and $\alpha = (s', v)$. When x is a static variable, the successor is $q = (h, g', \gamma)$ where $g' = g[x \mapsto d]$. When x is only a local variable in that FSM, the successor is $q = (h, g, \gamma')$ with $\gamma' = \alpha\beta$ and $\alpha = (s', v[x \mapsto d])$.

Examples of these rules are given in the next section on the Nim game, together with the model transformation step.

4.3. PDS model of the Nim game

Figure 10 shows a part of the PDS of the Nim game. It illustrates the application of the transition rules presented in Section 4.2. In this figure, there are a set of configurations (rounded boxes separated by arrows). Each configuration is composed of a heap state (leftmost inner box, h), a global variable state (middle inner box, g) and a stack state (rightmost inner box, γ). The PDS evolves from top to bottom.

The Nim game has two constants: `NBRow` and `MaxNBtaken` which are transformed into literal values during the procedure from VeriJ to HFSM. It does not have any other static variables, hence the global variable is reduced to $\emptyset$ in this case. Whenever an instantiation happens with the `new` operator, an object is added to the heap state. Its fields hold the default values of their respective types.

The main method of Nim game contains the object creation $\delta : Nim\ nim = new\ Nim()$;. In the PDS, it is decomposed into three steps: (*i*) reference variable declaration $Nim\ nim$, (*ii*) instantiation $nim = new\ Nim$ and (*iii*) initialization by invoking the constructor $nim.Nim()$. This pushes the initial state of Nim onto the stack ($Nim\ S.ini, [this \mapsto \$1]$ in 4^{th} configuration).

In the constructor declaration $Nim()$, the class instance creation statement $this.board = new\ Board()$; is decomposed into two steps: $this.board = new\ Board$ and $board.Board()$. This pushes the initial state of Board onto the stack ($Board\ S.ini, [this \mapsto \$2]$ in 6^{th} configuration).

Due to the lack of space, we skipped configurations related to further statements up to the return of the call to the constructor of Board. In the 7^{th} configuration, the final state $Nim\ S1$ is reached. Then, upon the return of the call to constructor of Nim, the topmost stack symbol is popped, which leads to the 8^{th} configuration.

5. CONCLUSION AND PERSPECTIVES

This paper presents VeriJ, a language for the modeling and controller synthesis dedicated to complex systems. Based on a limited subset of Java, VeriJ also contains specific elements for the purpose of solving controllability. This approach combines the advantages of an easy specification with Java, including the facilities of an integrated development environment, with the use of existing verification tools, acting on standard transition systems.

Compilation of VeriJ specifications into transition systems, as well as a preprocessing step, are performed with a complete chain of model to model transformations, using state-of-the-art model-driven engineering frameworks like Modisco and ATL. These operations (`Java2VeriJ.atl` and `VeriJ2Hfsm.atl`) produce, from a program, a set of hierarchically structured finite automata, interpreted using pushdown system semantics, hence enabling the use of recursion.

Contrary to software model-checking, where a large scope of programs is targeted, we focus on model generation, with controller synthesis as primary goal. Our approach is well suited to software engineers or domain experts wishing to use existing tools for controller synthesis.

Future steps of this work include checking the correctness of the two sets of ATL rules as recently done in (Planas et al. 2011; Ehrig and Ermel 2008) and linking this model generation with the controller synthesis:

1. Final encoding of a pushdown system into Hierarchical Decision Diagrams (SDD) (Thierry-Mieg et al. 2009), an efficient symbolic data structure used by our verification tool, `libddd`[7].
2. Extending the early experiments of a controller synthesis engine, based on this tool, which

[7] `http://ddd.lip6.fr`

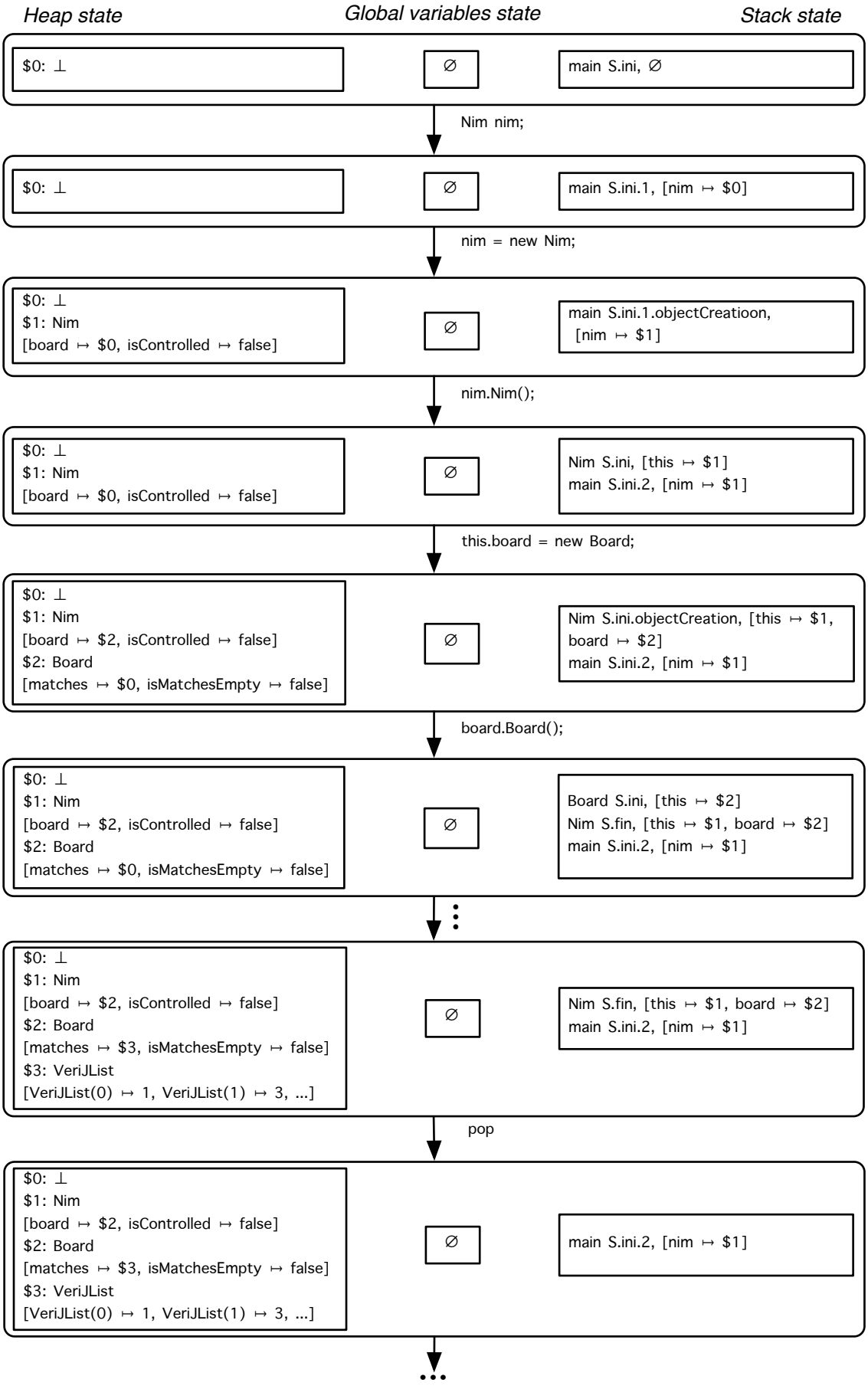


Figure 10: *A part of the PDS of Nim game.*

showed promising results in terms of scalability (Zhang et al. 2010).

A further goal is to apply this technique on industrial size complex systems, thus completing the centralized control of an automated highway system initiated in (Bérard et al. 2008). We expect this approach to be part of an industrial modeling, verification and control synthesis tool kit to handle complex systems specifications.

6. REFERENCES

Bérard, B., Haddad, S., Hillah, L., Kordon, F., and Thierry-Mieg, Y. (2008). Collision Avoidance in Intelligent Transport Systems: towards an Application of Control Theory. In *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES'08)*, pages 346–351, Göteborg, Sweden. IEEE Press.

Bouton, C. L. (1901). Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3:35–39.

Brat, G., Havelund, K., Park, S., and Visser, W. (2000). Java PathFinder - Second Generation of a Java Model Checker. In *Proceedings of the Workshop on Advances in Verification*.

Chaki, S., Clarke, E. M., Kidd, N., Reps, T. W., and Touili, T. (2006). Verifying Concurrent Message-Passing C Programs with Recursive Calls. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 334–349.

Chen, H. and Wagner, D. (2002). MOPS: an Infrastructure for Examining Security Properties of Software. In *In Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244. ACM Press.

Chomsky, N. (1962). Context-free grammars and pushdown storage. In *Quarterly Progress Report No. 65*, pages 187–194. MIT Research Lab. Elect., Cambridge, Mass.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., and Zheng, H. (2000). Bandera: Extracting Finite-state Models from Java Source Code. In *22nd International Conference on Software Engineering*, pages 439–448. ACM Press.

Ehrig, H. and Ermel, C. (2008). Semantical correctness and completeness of model transformations using graph and rule transformation. In *Proceedings of the 4th international conference on Graph Transformations*, ICGT '08, pages 194–210, Berlin, Heidelberg. Springer-Verlag.

Gvero, T., Gligoric, M., Lauterburg, S., d'Amorim, M., Marinov, D., and Khurshid, S. (2008). State extensions for Java PathFinder. In *Proceedings of the 30th international conference on Software Engineering (ICSE'08)*, pages 863–866, New York, NY, USA. ACM.

Havelund, K. (1999). Java PathFinder, A Translator from Java to Promela. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 152–, London, UK. Springer-Verlag.

Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295.

Jouault, F. (2005). Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pages 29–37.

Oettinger, A. G. (1961). Automatic syntactic analysis and the pushdown store. In *Structure of Language and Its Mathematical Aspects*, volume 12 of *Symposia on Applied Mathematics*, pages 104–129. American Mathematical Society.

Planas, E., Cabot, J., and Gómez, C. (2011). Two Basic Correctness Properties for ATL Transformations: Executability and Coverage. In *Proceedings of the 3rd International Workshop on Model Transformation with ATL*, pages 1–9, Zürich, Switzerland.

Ramadge, P. and Wonham, W. (1987). Supervisory Control of a Class of Discrete-Event Processes. *SIAM Journal of Control and Optimization*, 25(1):206–230.

Suwimonteerabuth, D., Berger, F., Schwoon, S., and Esparza, J. (2007). jMoped: a test environment for java programs. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 164–167, Berlin, Heidelberg. Springer-Verlag.

Thierry-Mieg, Y., Poitrenaud, D., Hamez, A., and Kordon, F. (2009). Hierarchical Set Decision Diagrams and Regular Models. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th Int. Conference, TACAS 2009*, volume 5505 of *LNCS*, pages 1–15. Springer.

Zhang, Y. (2010). Modeling Automated Highway Systems with VeriJ. In *MOdelling and VErifying parallel Processes (MOVEP)*, pages 138–143.

Zhang, Y., Bérard, B., Kordon, F., and Thierry-Mieg, Y. (2010). Automated Controllability and Synthesis with Hierarchical Set Decision Diagrams. In *Proceedings of the 10th International Workshop on Discrete Event Systems (WODES'10)*, pages 291–296.

Ziller, R. (2002). Finding Bad States during Symbolic Supervisor Synthesis. In Ruf, J., editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 209–218, TÃbingen, Germany. Shaker.