Controllability for discrete event systems modelled in VeriJ

Yan Zhang* and Béatrice Bérard*

Université Pierre and Marie Curie,
CNRS-UMR 7606 (LIP6/MoVe),
4 Place Jussieu, F-75005 Paris, France
E-mail: Yan.Zhang@lip6.fr
E-mail: Beatrice.Berard@lip6.fr
*Corresponding authors

Lom Messan Hillah

Université Pierre and Marie Curie,
CNRS-UMR 7606 (LIP6/MoVe),
4 Place Jussieu, F-75005 Paris, France
and
Université Paris Ouest Nanterre La Défense,
200, Avenue de la République,
F-92001 Nanterre Cedex, France
E-mail: Lom-Messan.Hillah@lip6.fr

Fabrice Kordon and Yann Thierry-Mieg

Université Pierre and Marie Curie,
CNRS-UMR 7606 (LIP6/MoVe),
4 Place Jussieu, F-75005 Paris, France
E-mail: Fabrice.Kordon@lip6.fr
E-mail: Yann.Thierry-Mieg@lip6.fr

Abstract: Existing tools for controllability checking mostly apply to abstract formalisms like finite automata or Petri nets. To avoid costly building of low-level formal models for large complex systems, we propose a programming language called VeriJ, a subset of Java with additional constructs dedicated to supervisory control, to model these systems in a familiar and friendly development environment. We provide a prototype tool chain, based on model transformation and pushdown automata, to automatically transform a system described in VeriJ into a labelled transition system (LTS). A controllability engine for this LTS is then integrated to the tool. To limit the state space explosion problem, we also add several mechanisms including garbage collection, abstraction, state compression, and partial exploration. Our approach, illustrated with a VeriJ model of the Nim game, shows that it is possible to combine: 1) the benefits resulting from using mature Java development environments; 2) performances comparable to those of existing tools.

Keywords: VeriJ; Java; model transformation; verification; controllability; discrete event systems; critical systems.

Reference to this paper should be made as follows: Zhang, Y., Bérard, B., Hillah, L.M., Kordon, F. and Thierry-Mieg, Y. (2014) 'Controllability for discrete event systems modelled in VeriJ', *Int. J. Critical Computer-Based Systems*, Vol. 5, Nos. 3/4, pp.218–240.

Biographical notes: Yan Zhang is a PhD student in Computer Science at the University Pierre and Marie Curie (UPMC) and member of the LIP6 research laboratory. Her areas of interest are formal verification techniques including modelling complex systems and controller synthesis.

Béatrice Bérard is a Professor of Computer Science at UPMC and member of LIP6 since 2008. Her main research interests are quantitative verification and synthesis.

Lom Messan Hillah is an Associate Professor at University Paris Ouest Nanterre and member of LIP6 since 2010. His main research interests lie in the integration of formal methods in industrial software development methodologies.

Fabrice Kordon is a Professor of Computer Science at UPMC and member of LIP6 since 1996. His main research interests lie in distributed systems, software engineering and formal methods.

Yann Thierry-Mieg is an Associate Professor at UPMC and member of LIP6 since 2005. His main research interests are efficient algorithms and data structures for verification.

This paper is a revised and expanded version of a paper entitled 'Modeling complex systems with VeriJ' presented at the 5th International Workshop on Verification and Evaluation of Computer and Communication System (VECOS), Tunis, Tunisia, 15 September 2011.

1 Introduction

1.1 Context

Given a discrete system model $\mathcal{M}$ 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 seen as a two-player game, *environment* versus *controller*, where controllability corresponds to the existence of a winning strategy for the *controller* to satisfy φ, against all possible behaviours of the *environment*. In case of a positive answer, the final goal is to generate a controller or, equivalently, a winning strategy. We consider here safety objectives, where formula φ expresses avoidance of undesirable situations.

The control problem has been largely studied since the work of Ramadge and Wonham (1987) and several tools have been developed (Behrmann et al., 2007; Moor et al., 2010). Two main obstacles limit practical application of these techniques in industry:

a building low-level formal models (such as automata) is often considered too costly, and

b due to the state space explosion occurring for large systems, scalability is usually a major issue.

Aiming at solving problem (a), we previously defined a language called VeriJ (Zhang, 2010) to model complex systems.

1.2 Contribution

In this paper, we propose an approach to perform controllability checking on a labelled transition system (LTS) generated from a VeriJ program:

- our first contribution is the completed transformation chain, from VeriJ source code to LTS (this contribution is an extension of Zhang et al., 2011)
- we further implement several mechanisms to address the state space explosion problem: garbage collection, abstraction, state compression (SC) and bounded exploration
- we also provide controllability checking for VeriJ programs, as a first step for controller synthesis.

We use here the Nim game as a running (scalable) example. Experiments are provided also on a larger example dealing with automated highway control.

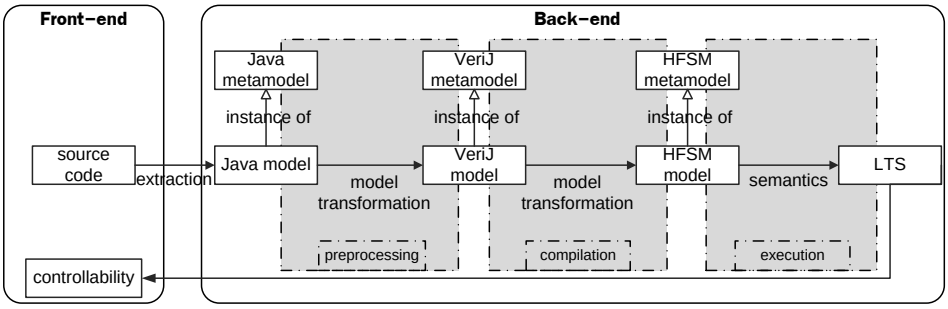
1.3 Outline

Section 2 briefly recalls the controllability algorithm, introduces VeriJ and gives an overview of the transformation chain from a VeriJ program to an LTS. In Sections 3 and 4, we describe the details of this transformation. Techniques to reduce the state space and the exploration are given in Section 5 and we compare our approach with other verification tools in Section 6.

2 Overview of the VeriJ approach

Our target is to provide a tool consisting of a front-end where users can model a system by programming it in VeriJ, together with a back-end that generates a LTS from the VeriJ program, and answers whether the input system is controllable or not. The global view is given in Figure 1 and described in more detail in the rest of the sections. We first present the classical framework of controllability.

Figure 1 From source code to formal model and controllability test



2.1 Controllability

A (non-deterministic) LTS over a set Act of actions is a triple $\mathcal{M} = (S, s_0, \rightarrow)$ where S is a set of configurations (or states), $s_0 \in S$ is the initial configuration and $\rightarrow \subseteq S \times Act \times S$ is the transition relation. The set of actions is partitioned into *environment* and *controller* actions. The set $S_{reach} \subseteq S$ of reachable states contains all states $s \in S$ for which there exists a path in the graph from s_0 to s.

Let S_{fail} be a subset of S_{reach} containing states to be avoided. The controllability problem is to check if there exists a strategy of the controller to avoid S_{fail} against all environment behaviours. The classical algorithm (Ramadge and Wonham, 1987) consists in a backward exploration starting from S_{fail} and adding new states to this set as follows:

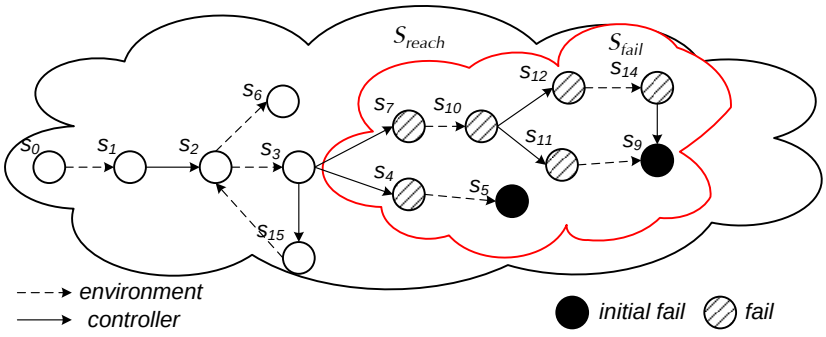
1 any state from which the environment can reach a failure state in a single move is added to S_{fail}

2 any state from which all controller moves lead to a failure state is added to S_{fail}.

When a fixpoint is reached, either the initial state is a failure state, indicating that the system is not controllable, or there exists a winning strategy for the controller.

Figure 2 illustrates this setting, with environment actions as dashed arrows. The initial failure states are black (e.g., s_5) and those added by the algorithm are striped (e.g., s_4 or s_{10}). The system depicted in this figure is controllable since s_0 does not belong to S_{fail} when the algorithm terminates.

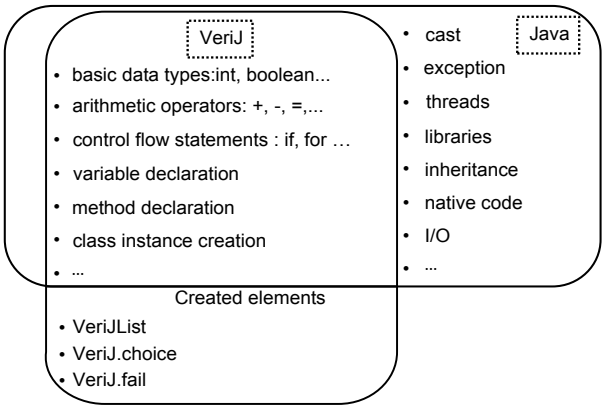
Figure 2 Illustration of the controllability algorithm (see online version for colours)



2.2 VeriJ syntax

VeriJ is inspired by Java for the expressiveness and simplicity to model complex systems. It contains a core subset of Java and includes in addition several constructs for supervisory control (see Figure 3).

Figure 3 Relation between VeriJ and Java



2.2.1 Java subset

The syntax taken from the Java language includes basic data types, integer arithmetic, object creation and instanciation, method invocation, assignments, and standard conditional and control statements. However, VeriJ is simplified with respect to Java since its purpose is to model systems, not to implement them. Currently, VeriJ does not support features like exceptions, type casting, inheritance, libraries or native code. Concurrency is an important feature of complex systems but it is not yet implemented and it is left out in this paper.

`VeriJList.` Large systems may involve components and handle, for instance, dynamic lists. In Java, even the simplest collection implementation (`ArrayList`) contains several hundred lines. Due to this complexity, we create instead a single `VeriJList` type, to handle basic collections with a small set of operations, hence providing a high level of abstraction for collections.

2.2.2 Control elements

In VeriJ, two methods can be called to carry the necessary information:

- `VeriJ.choice` models an arbitrary choice among possible actions at a decision point. More precisely, the user can call the method `int VeriJ.choice(int min, int max, int playerID, int actionID)`, where parameters `min` and `max` give the bounds of an interval of integers representing action indices, `playerID` identifies by whom the choice is made and `actionID` indicates which action type is chosen.
- `VeriJ.fail()` can be called to designate a failure state.

Example of modelling using VeriJ. We illustrate the syntax of VeriJ with the Nim game. A set of matches are arranged on a board in several rows, with $2i + 1$ matches in row i (rows are numbered from 0). The game is played as follows: in turn, each player selects one of the non-empty rows and chooses a number of matches to remove from it. The player who takes the last match (or matches) looses. This game was completely solved by Bouton (1901). A winning strategy exists if the player can reach a safe position where the *exclusive or* of the binary encoding of the number of matches in each row is equal to zero. We can check the correctness of our controllability procedure with respect to this solution.

The corresponding VeriJ program is described in Figure 4. Parameters of the system are defined in class `Constants` (ℓ.2-6). The set of matches is initialised as a VeriJList of integers (ℓ.11). Player choices of row and number of matches to take are defined by the calls to `VeriJ.choice` (ℓ.16-19). The main function contains the simulation loop (ℓ.27-33), where failure states are produced when the controller takes the last remaining matches (ℓ.31-33). We can let the controller start the game by exchanging ℓ.28-30 and ℓ.31-33 without changing failure states definition.

Figure 4 Nim source code (in separate .java classes) (see online version for colours)

```
1 public class Constants {
2   public static final int ENV = 0;      // ENVIRONMENT
3   public static final int CONTR = 1;    // CONTROLLER
4   public static final int NBRow = 3;    // number of rows
5   public static final int CHOOSE_ROW = 0; // action IDs
6   public static final int CHOOSE_MATCH = 1;
7 }
8 public class Board {
9   VeriJList<Integer> matches;
10  public Board() {
11    this.matches = new VeriJList<Integer>();
12    for (int i = 0; i < Constants.NBRow; i = i + 1) {
13        int num = 2 * i + 1;
14        matches.add(num);      }}
15  public void playMove(int id) {
16    int row = VeriJ.choice(0, matches.size()-1,
17              id, Constants.CHOOSE_ROW);
18    int nb = VeriJ.choice(1, matches.get(row),
19              id, Constants.CHOOSE_MATCH);
20    matches.set(row, matches.get(row)-nb);
21    if (matches.get(row)==0){
22      matches.remove(row); }}
23 }
24 public class Nim {
25  public void main(String[] args){
26    Board board = new Board();
27    while(true){
28      board.playMove(Constants.ENV);  // environment plays
29      if(matches.size()==0)
30        return;
31      board.playMove(Constants.CONTR); // controller plays
32      if(matches.size()==0) //controller loses
33        VeriJ.fail(); }}
34 }
```

Preprocessing step. In this step (see Figure 1), we use MoDisco (http://www.eclipse.org/MoDisco) to extract, from the Java/VeriJ source code, a model of the application, conforming to the Java metamodel. This Java model is transformed into a model conforming to the VeriJ metamodel, by pruning unnecessary information from the Java model and building the VeriJ specific elements. This step is implemented as a model to model transformation using the Atlas Transformation Language (ATL, http://www.eclipse.org/atl), a state-of-the-art model transformation plugin within Eclipse.

Any programmer can quickly get started with the tool since the syntax is close to Java. Although the model description is textual (a program), some tools provide multiple forms of visualisations (UML class diagrams, browsers, ...) if a graphical representation is desired. We also provide adapters that implement the additional features (`ArrayList` for `VeriJList`, the standard Java `random` function for `VeriJ.choice` and a system exit or error for `VeriJ.fail()`). Then, the user can simulate the system by executing the program.

2.3 VeriJ semantics

Analysing a VeriJ specification requires to build an LTS representing all program executions. Two alternative solutions have been proposed for such program transformation.

2.3.1 Direct translation

This approach aims at translating input source code into the representations accepted by existing verification tools. This approach was used for instance in early versions of JPF (Havelund, 1999) which transform Java to Promela. The latter is the input language of SPIN model checker (http://www.spinroot.com) based on communicating finite automata. It was also the case for Corbett et al. (2000) which, using slicing and abstraction, translates the input source code into input of some existing finite state machine (FSM) verification tools (i.e.,, SPIN, SMV, etc.) through Bandera Intermediate Representation.

Although reusing these tools is then possible, some features of the language can be difficult to translate (like inheritance or object creation).

2.3.2 Compilation

The second option consists in compiling the source code, then generating a formal model from the result. With this option, there are two main branches according to different intermediate languages:

1 a standard intermediate language, for instance, Java bytecode for Java or assembly language for C

2 a control flow graph (CFG) reflecting the program structure as the intermediate language.

1 *Standard compilation.* This approach uses a standard compiler to derive the semantics of the source code, and handle the verification on transition systems by executing the Java bytecode (or assembly language for C). It solves issues related to software artifacts, such as external libraries for which no source code is available, hence it is the preferred option for full-fledged software model-checkers. It also produces less cases when implementing the verification tool as the variety of opcodes is rather limited. This is the choice taken in recent versions of Java Path Finder (JPF, http://babelfish.arc.nasa.gov/trac/jpf) a software model-checker for Java which relies on a dedicated backtrackable Java Virtual Machine (JVM) that provides non-deterministic choices and control over thread scheduling. 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.

2 *Compilation to CFGs.* In this case, the program is transformed into a CFG that describes all possible execution paths. This approach preserves a high level of abstraction in the resulting system model. However, it may also be difficult in general to capture all syntactic elements from the language.

The interpretation of the CFG produces the final transition system. At this step, a straightforward approach consists in inlining all calls, resulting in a single FSM for the whole input program. FSMs are the natural input language for many model-checkers, which makes this solution attractive. For instance, both verification tools F-Soft (Ivančić et al., 2008) and BLAST (Beyer et al., 2007) transform C program into CFG first. However, the FSM obtained by inlining may contain redundant information due to duplication of behaviours (corresponding to function calls). Moreover, it cannot be applied when unbounded recursion is involved because the inlining could produce infinite structures.

Another popular approach is to use pushdown semantics to interpret the CFG. Pushdown automata, introduced in Oettinger (1961) and Chomsky (1962), are a natural choice for modelling method calls and interprocedural program behaviours, by adding a (possibly unbounded) stack to a finite set of control states. This produces a compact and accurate representation of procedure calls and it is the choice taken in several recent verification tools, for instance, jMoped (http://www7.in.tum.de/tools/jmoped), BLAST, SLAM (Ball et al., 2011) and in our own approach as well.

2.3.3 Our selection

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. On the contrary, CFG provides a graph notation to describe all possible paths that a program may traverse, which is close to formal models representation.

Thus, we choose the compilation-execution approach with compilation to CFG. We first transform a VeriJ program into hierarchical finite state machine (shortly HFSM), a variant of CFG that preserves the structure of the source code at a high level of abstraction. This corresponds to the *compilation* step in Figure 1 (details in Section 3). We then define specific pushdown rules to generate the final LTS on which analysis can be performed, in the *execution* step (details are provided in Section 4).

3 Generating HFSM from VeriJ

3.1 Hierarchical finite state machines

A FSM is a finite transition system equipped with final states. For our purpose, we define an FSM over a finite alphabet Σ as a tuple $F = (Q, \delta, q_0, q_f)$ where Q is a finite non-empty set of *states*, δ is a partial mapping from $Q \times \Sigma$ to Q, $q_0 \in Q$ is the *initial state* and q_f is a unique *final state*.

Definition 1: Let R and Σ be two finite disjoint alphabets and write $\hat{\Sigma} = \Sigma \uplus R$. An abstract HFSM is a finite set of FSMs $\mathcal{F} = \{F_r, r \in R \cup \{0\}\}$ over the alphabet $\hat{\Sigma}$, where 0 is a symbol not in R. F_0 is called the initial FSM of $\mathcal{F}$.

For any $r \in R \cup \{0\}$, we set $F_r = (Q_r, \hat{\Sigma}, \delta_r, q_{0,r}, q_{f,r})$. The set $Q = \uplus_{r \in R \cup \{0\}} Q_r$ (where $\uplus$ denotes the disjoint union) is called the set of states of $\mathcal{F}$ with $q_{0,0}$ the initial state and $q_{f,0}$ the final state of $\mathcal{F}$. The set $\Delta = \uplus_{r \in R \cup \{0\}} \delta_r$ is called the set of transitions of $\mathcal{F}$. A transition $t = q \xrightarrow{\sigma} q'$ with $\sigma \in \hat{\Sigma}$ from some $F \in \mathcal{F}$ is an internal transition of F if $\sigma \in \Sigma$ and a reference to FSM F_r if $\sigma = r \in R$.

With a VeriJ program we associate a program HFSM $\mathcal{H} = (\mathcal{F}, C, \lambda)$ where $\mathcal{F}$ is an abstract HFSM, C is a finite set of class names for the classes defined in the program, and $\lambda : \mathcal{F} \mapsto C$ is a (surjective but not necessarily injective) mapping associating a class name $\lambda(F) \in C$ with each FSM F in $\mathcal{F}$. In such an HFSM, F_0 corresponds to the `main` method and other FSMs correspond to functions called by the program executed from the `main` method. A transition within F corresponds to an instruction of the function and when labelled by a reference r, it corresponds to a call to the function associated with F_r.

When a VeriJ program contains game constructs, we consider two particular subsets of Q. The set Q_{fail} contains the states where `fail` is reached. The set Q_{choice} of choice states is: $Q_{choice} = \{q \in Q \mid q \xrightarrow{\texttt{choice}} q' \ is\ a\ transition\ in\ some\ F \in \mathcal{F}\}$.

A part of the HFSM corresponding to the Nim example of Figure 4 is depicted in Figure 5. It represents the two FSMs associated respectively with the method `playMove` of class `Board` and the `main` method of class `Nim`.

3.2 Transformation from VeriJ to HFSM

Our compilation of VeriJ (in Figure 1) is implemented by a model transformation into an HFSM. For this, we define a HFSM metamodel.

3.2.1 HFSM metamodel

Figure 6 shows the core part of the HFSM metamodel (the complete metamodel contains 58 metaclasses against 126 in Java).

A model is composed of a set of FSMs, each one consists of:

- a set of *State*s, including an initial state and a final state, each one having a single attribute *name*

- a set of *Transitions*, each of them has a source state, a destination state and a *TransitionExpression* denoting a simple statement (e.g., variable declaration, assignment, etc.) or an *HfsmExpression* (method invocation) referring to an FSM, hence bringing in the hierarchy
- a set of local variables representing parameters of a method declaration.

Figure 5 Extracts of the Nim HFSM model (see online version for colours)

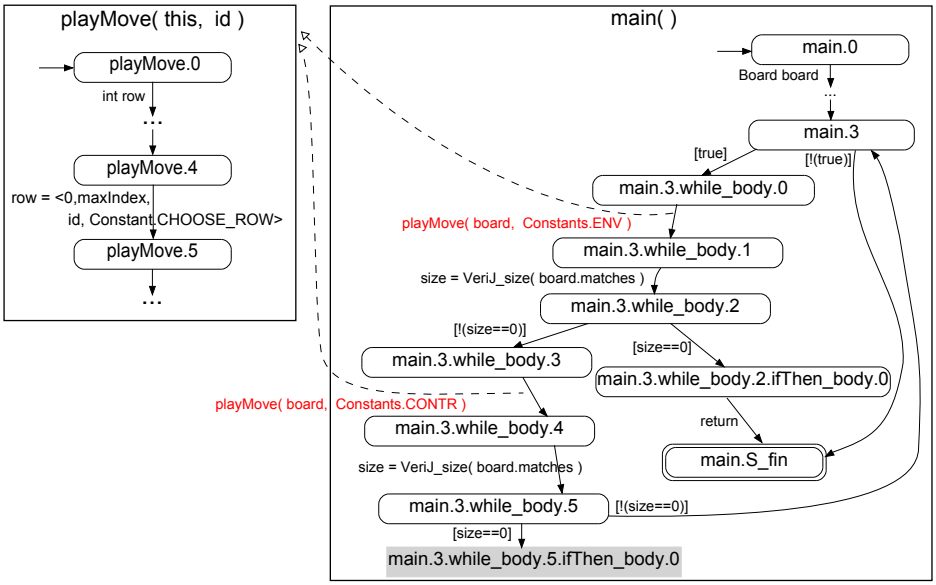
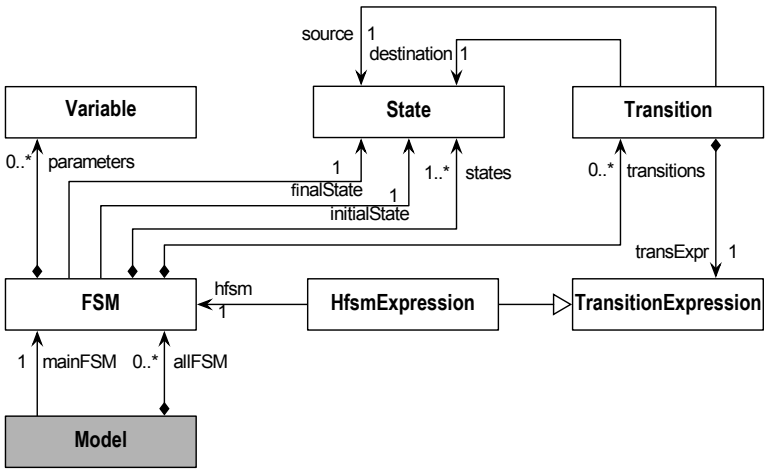


Figure 6 Core part of the HFSM metamodel



In the HFSM metamodel, both the types of parameters of a method declaration and the fields of a class belong to the *variable* metaclass. This element has three subclasses: IntVar, BoolVar and ObjectVar which represent respectively the three types integer,

Boolean and reference to an object. A Boolean tag is associated with variables to indicate if they are global or local.

There are eight kinds of basic transition types in HFSM:

1 *Variable declaration* restricted to int, Boolean, object reference and list variables.

2 *Object allocation* allocating memory in the heap for newly created objects.

3 *Assignment* of an expression, including arithmetic, VeriJList operations (for instance, `size`, `get`, `equals`), `VeriJ.choice`, etc., to a variable.

4 *Drop variables* to remove variables that fall out of scope.

5 *Return* to pop a context and possibly return a value.

6 *FSM method call* includes constructor and method calls. Member methods are handled as static methods, with `this` as additional parameter.

7 *VeriJ method call* mainly used to carry the VeriJList operations without return value: `set`, `add`, `remove`, `insert`.

8 *Boolean guard* for conditions in control flow statements.

3.2.2 Transformation

Using VeriJ and HFSM metamodels, we coded a set of ATL rules to create states and transitions for each FSM in this HFSM model. Each method in the source code is transformed into an FSM with the same name.

States are created according to the positions of atomic statements in the program. Each FSM F has initial and final states called respectively F.0 and F.S_fin. The unique final state can be the destination state for several transitions (for instance, *return*). To avoid duplication, states are named as follows: 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 instance, in Figure 5, main.3.whilebody.3 represents the source state of the method call at line 31.

Each transition is obtained from a statement of the VeriJ model. It takes the name of the atomic statement as the label *transExpr*, with created source and destination states. If a transition is a method call, its label refers to the FSM of the method. For instance, the method invocation of `playMove(board, Constants.ENV)` references the FSM on the left of Figure 5.

For the `VeriJ.choice` method, an assignment of the form `x = VeriJ.choice(min, max, playerID, actionID)` is transformed into x=⟨min, max, playerID, actionID⟩. Finally, any state involving a call to `VeriJ.fail()` is labelled as a failure state. Note that such a state has no successor. It is depicted in grey background, like for instance state main.3.while_body.5.ifThen_body.0.

Since some VeriJ statements have richer semantics than a single atomic HFSM transition, we need to split them into several atomic transitions. We give two examples below:

- *Instantiate a class* of the form: `ClassObj obj = new ClassObj();`

  Three steps are needed for this statement:

  1 declare a variable `obj` as a reference variable $\delta 1$: ClassObj obj;

  2 allocate memory for the new object in the heap and return a reference to that memory cell $\delta 2$: obj = new ClassObj;

  3 method invocation of the constructor call $\delta 3$: obj.ClassObj();.

  For instance, `Nim nim = new Nim();` is transformed into: $\delta 1$: Nim nim;, $\delta 2$: nim = new Nim; and $\delta 3$: nim.Nim();.

- *Method call returning a value* of the form `var = m(arg1,arg2 ... );`

  For this statement, we need an additional variable returnVar typed `varType` (the same as `var`). The statement is split into four atomic updates: declaration of `returnVar` $\delta 1$: varType returnVar; method call $\delta 2$: m(arg1, arg2 ...); where the value to return will be assigned to `returnVar`, assignment of the result to the caller variable $\delta 3$: var = returnVar; and dropping the temporary variable $\delta 4$: drop returnVar;.

4 Generating LTS from HFSM

In this section, we show how to execute the HFSM as a pushdown system (step called *execution* in Figure 1) and add special labels for controllability. We first recall the classical model of pushdown automata (Oettinger, 1961; Chomsky, 1962).

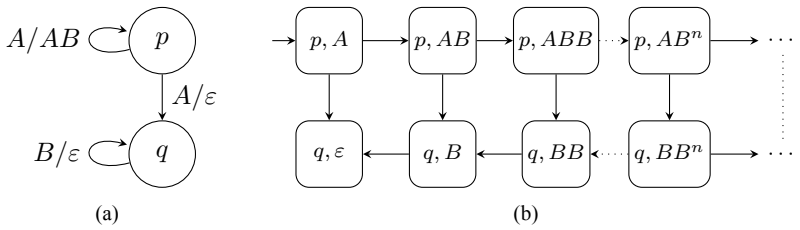
4.1 Pushdown automata

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 $s_0 \in P \times \Gamma$ 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 an LTS with $S = P \times \Gamma^*$, the set of configurations. For a rule $\delta : (p, z) \hookrightarrow (p', \alpha)$ in Δ 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 7 shows an example of a PDA with its semantics. In Figure 7(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 $s_0 = (p, A)$. The PDA has an infinite set of configurations depicted in Figure 7(b).

Figure 7 A PDA example, (a) a PDA $\mathcal{P}$ (b) transition system of $\mathcal{P}$



4.2 Variables in VeriJ programs

Let X be a set of variables, including *this*. For each $x \in X$, let D_x be the 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*. The default value (0 for `int`, `false` for `Boolean` or $\$0$ for *Ref*) is noted $\perp$ for all ranges.

A valuation is a partial mapping $v : X \to \cup_{x \in X} D_x$ 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 $[-]$ the valuation such that $Dom(v) = \emptyset$.

For a valuation v, $y \in X$ and $d \in D_y$, we 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 $v' = v[y \mapsto d]$. 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 $i = 1, \ldots, n$.

4.3 LTS of a VeriJ program

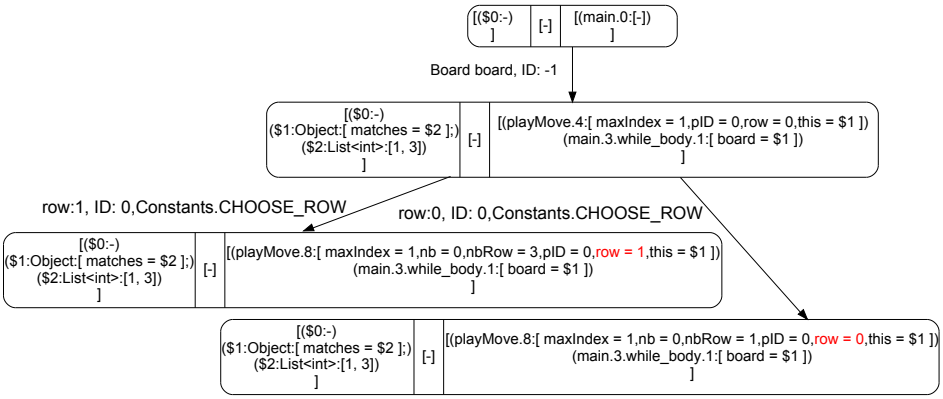
Given a program HFSM $\mathcal{H} = (\mathcal{F}, C, \lambda)$ with initial FSM F_0 and set of states Q, we denote by V the set of all valuations for the program variables. We define the stack alphabet by $\Gamma = Q \times V$.

The set of configurations of the LTS is defined by: $S = (C \times V)^* \times V \times \Gamma^*$. Hence, a configuration $s = (h, g, \gamma) \in S$, consists of:

- $h \in (C \times V)^*$ the *heap state*. Hence, an empty heap is described by ε (also represented in the Figure 8 by $\$0 : \perp$). 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, so a non-empty heap is written as $h = (c_1, w_1) \ldots (c_n, w_n)$ for some $n \geq 1$. Adding a new object to h is written $h.(c, w)$ for some (c, w).
- $g \in V$ the *global variable state* is the valuation of static variables and the values of 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 $(q, v) \in \Gamma$ is composed of an FSM state and a variable valuation.

The initial configuration is $s_0 = (\varepsilon, [-], (main_0, [-]), 0)$, where $main_0$ denotes the initial state of F_0.

Figure 8 Part of the LTS generated for the Nim game after abstraction (see online version for colours)



We now define the transition relation $\rightarrow$ of the LTS. A transition from configuration $s = (h, g, \gamma)$ to configuration $s' = (h', g', \gamma')$ is written $s \xrightarrow{t} s'$, where $t : q \xrightarrow{\delta} q'$ is a transition from some 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 correspond to method invocation: stacking the initial state of the method called and the return address after popping the topmost stack symbol. If $q \in Q_{fail}$ in the HFSM, then the configuration is also a failure.

Below are presented several examples of transitions corresponding to the eight types of HFSM transitions ($t : q \xrightarrow{\delta} q'$) described in the previous section. In configuration $s = (h, g, \gamma)$, we assume that the heap size is $|h| = n$ and the stack is $\gamma = z\beta$, with $z = (q, v)$ the topmost symbol.

1 *Variable declaration* of the form δ : type x = initialiser;

If x is local, this transition adds to v a valuation for x, assigning the result d of the evaluation of $initialiser$. The successor configuration is $s' = (h, g, \gamma')$, where $\gamma' = \alpha\beta$ with $\alpha = (q', v \cup [x \mapsto d])$. If x is global, g is changed into $g' = g \cup [x \mapsto d]$ and $\alpha = (q', v)$.

2 *Object allocation* of the form δ : x = new Class;

For a variable x declared as a reference variable, this operation allocates memory for the new object in the heap and returns a reference to that memory cell. The successor configuration is $s' = (h', g, \gamma')$ where $\gamma' = \alpha\beta$ with $\alpha = (q', v \cup [x \mapsto \$(n+1)])$. If Class is not VeriJList, the new heap state is $h' = h.(c, w)$, where $c =$ Class and w assigns default values to all fields. Instantaneously, another rule (*Method call*) will be applied for the constructor call. For VeriJList, $h' = h.(t, (-))$, where t is the type of the list elements and $(-)$ is an empty valuation.

3 *Assignment* of the form δ : x = expression;

According to different types of x (field or local variable), and of expression types, the configuration updates the variable valuation in different parts in a natural way. For example, assume a local variable is assigned by a VeriJList operation of the form σ : x = li.m(arg$_1$, arg$_2$...);. Then x takes the value m(arg$_1$, arg$_2$...) of the object li. If this value is d, the successor state is $s' = (h, g, \gamma')$ with $\gamma' = \alpha\beta$ and $\alpha = (q', v[x \mapsto d])$.

If a non-deterministic choice is assigned to a local variable by x = ⟨min,max,playerID,actionID⟩, this transition has a successor $s'_i = (h, g, \gamma'_i)$ for each $i \in$ [min,max], with $\gamma'_i = \alpha_i\beta$ and $\alpha_i = (q', v[x \mapsto i])$.

4 *Drop variable* of the form δ : drop x;

Let d be the value of x. If x is a local variable, let $z = (q, v_1 \cup [x \mapsto d])$ be the topmost stack symbol, the successor state is $s' = (h, g, \gamma')$ with $\gamma' = \alpha\beta$ and $\alpha = (q', v_1)$. If x is a global variable, with $g = v_1 \cup [x \mapsto d]$, the successor is $s' = (h, g', \gamma)$ with $g' = v_1$.

5 *Return* of the form δ : return expression;

With this HFSM transition, leading to the the final state of an FSM, the topmost symbol is popped from the stack. The successor state is $s' = (h, g, \beta)$ when expression is empty. Otherwise, let d be the value of expression, then $s' = (h, g', \beta)$ with $g' = g[\mathit{returnVar} \mapsto d]$. Notice that, reaching the final state without a return statement also pops the topmost symbol hence producing $s' = (h, g, \beta)$.

6 *FSM method call* of the form σ : m(ob, arg$_1$, arg$_2$ ...);

Let m_0 be the initial state of the FSM associated with method `m()`, and let ob, x_1, x_2... be the parameters. The successor state is $s' = (h, g, \gamma')$ with $\gamma' = \alpha\beta$ and $\alpha = ((m_0, v_0)(q', v))$, where v_0 is the valuation defined by $v_0(x_i) = arg_i$ for $i = 1, 2, \ldots$ and $v_0(this) = \$j$, where $\$j$ is the reference of object ob.

7 *VeriJ method call* of the form δ : m(li, arg$_1$, arg$_2$ ...);

This HFSM transition is an operation of the elements of VeriJList li. For a heap state $h = (c_1, w_1)...(c_n, w_n)$, if $li \mapsto \$i$, for some i ($1 \leq i \leq n$), the valuation is written as $w_i = [li(0) \mapsto d_0, ..., li(m) \mapsto d_m]$, where m is the index of the last element. The successor state is $s' = (h', g, \gamma')$ with $\gamma' = \alpha\beta$ and $\alpha = (q', v)$. The heap state h' depends on the operation. For instance, for li.add(d), $h' = (c_1, w_1)...(c_i, w'_i)...(c_n, w_n)$, where $w'_i = w_i \cup [li(m+1) \mapsto d]$.

8 *Boolean guard* of the form δ : [expression]

This HFSM transition label, as well as [!expression], are the conditions of two branches in a control flow statement. If expression is evaluated to `true`, the successor state is $s' = (h, g, \gamma')$ with $\gamma' = \alpha\beta$ and $\alpha = (q', v)$. Otherwise, the other branch is taken.

Figure 8 depicts a (small) part of the LTS obtained for the Nim program and illustrates transition rule (3) assigning a non-deterministic choice to a local variable at playMove.4. Application of the rules above yields a much larger LTS [see (Zhang et al., 2011) for an example]. This figure is obtained after performing the abstraction presented in Section 5). In this figure, each configuration (depicted as a large rounded box) has a heap state (leftmost inner box, h), a global variable state (middle inner box, g) and a stack state (rightmost inner box, γ). The Nim game has no static variable except the five constants which are transformed into literal values during the procedure from VeriJ to HFSM. Hence the global variable state g is reduced to $[-]$ in this case. The first box presents the initial configuration s_0. The second box is the source state of the choice, where the heap h shows the instantiation of the `Board`: an object $matches$ refers to a list at address \$2 with integer values. The transition produces two successors with different values of local variable row in the topmost symbol of the stack state. The relevant information of the transition is extracted in the transition label.

5 LTS state space reductions

To cope with the very large LTSs generated from VeriJ programs, we need to reduce the state space size. Many related works are dedicated to such reduction: symbolic verification (Burch et al., 1992), predicate abstraction (Flanagan and Qadeer, 2002), partial order reduction (Bogdoll et al., 2011), symmetry reduction (Sistla and Godefroid, 2004), on-the-fly verification (Lime et al., 2009), state compression (Laarman et al., 2011), etc. To have better performances, these techniques can be combined.

5.1 Reductions

Our prototype implements garbage collection, abstraction of irrelevant states (with respect to a given criterion), SC and partial exploration. Instead of using the Nim game, which is a toy example, we experimented these techniques on an industrial case, consisting of a section of an automated highway system (Bérard et al., 2008) (see Table 1). In this system, the controller's goal is to avoid collisions, while the environment can introduce new cars and control at most one car.

5.1.1 Garbage collection

Since VeriJ has no explicit destruction of objects allocated in the heap, unreferenced objects are kept in the heap state. Moreover, since there is no predefined bound on the state evolution loop, any object creation in the loop produces an infinite structure. This loop appears for instance at line 27 in Figure 4, although it contains no object creation in the case of the Nim game.

Garbage collection (GC) discards unreferenced objects in the heap and then compacts the heap to avoid the resulting fragmentation. We use a mark-and-sweep algorithm on every relevant configuration (which consists in marking unreferenced objects and remove them).

5.1.2 Abstraction of the state space (Abs)

Due to explicit interpretation of program instructions, even small examples may yield intractable LTSs. We propose to use an abstraction criterion to avoid representing parts of the LTS. This criterion is defined as a Boolean predicate over configurations. During the generation of the LTS, from the current configuration s, any successor s' marked as abstract is inductively replaced by its set of successors. An abstraction criterion producing cycles of abstract states is considered ill-formed (the procedure would not terminate). This can be handled by bounding the successor's distance to some fixed k.

We define a criterion to abstract any state that

1 does not proceed a call to `VeriJ.choice`

2 is not initial or final or failure state. This produces an aggregation of deterministic paths in the LTS, leaving mainly the decision points.

5.1.3 State compression

To limit the memory requirements of the controllability analysis, we use a dedicated compression scheme inspired by Holzmann (1997) aiming at reducing the memory amount used per state. It consists in using several unique tables for state elements like variable valuations, to share the representation of common parts among states. We use a multilevel variant in which heap object states, global variable states and stack symbols share their memory representation, heap and stack are themselves unique in memory. This type of compression is also used in JPF, Spin and MoonWalker (http://fmt.cs.utwente.nl/tools/moonwalker).

5.1.4 Partial exploration

When the system parameters scale up, the combinatorial explosion problem may prevent full LTS generation. In such a case, we revert to bounded depth generation, with a breadth-first search algorithm. In other words, we partially explore the state space and check controllability on it.

If the subsystem is uncontrollable for some depth k, the environment can win in less than k moves, which allows to conclude that the system is not controllable. Conversely, a controllable subsystem does not guarantee that the whole system is controllable. This technique can be useful for further study in controller synthesis and is not experimented here.

5.2 Experiments

They are conducted on the highway example mentioned above, with n lanes, total length L, and minimum safety distance d_{min}. The source code contains 14 classes, 62 methods and more than 500 lines. More details (including partial exploration) can be found at http://pagesperso-systeme.lip6.fr/Yan.Zhang/example14.html.

All experiments in this paper were made on a 2.66 GHz Core2 Quad PC running Linux with 1 GB of memory. In Tables 1 and 2, the number of lanes is $n = 2$. And for

Table 1, the safe distance is $d_{min} = 3$. We use $-$ in the cell for any execution that does not meet the constraints (for instance, when more than 1 GB of memory is used).

Table 1 presents the memory and time used without optimisation and after implementing the first three techniques above. The unbounded evolution loop causes out of memory when no garbage collection is applied (first columns). Although GC permits to produce a finite state space (columns GC), it runs out of memory much earlier than in combination with the other techniques. Our specific abstraction (columns GC+Abs) decreases both the memory and time consumption. Generally speaking, the SC (columns GC+Abs+SC) takes slightly more time, but compacts the memory up to 51%.

Table 1 Memory (*MB*) and time (*s*) usage in the automated highway

L	*No optimisation*		*GC*		*GC + Abs*		*GC + Abs + SC*	
	Mem	*Time*	*Mem*	*Time*	*Mem*	*Time*	*Mem*	*Time*
6	-	-	167	2.9	126	1.6	125	1.7
7	-	-	638	25.2	360	5.7	324	6.2
8	-	-	-	-	371	8.6	331	9.3
10	-	-	-	-	440	20.3	350	20.8
11	-	-	-	-	997	168.3	504	165.9

6 Related work

Since VeriJ contains a core subset of Java and is dedicated to controllability, we compare our approach both with Java model checkers and supervisory control tools.

6.1 Comparison with software verification tools

Java model checking belongs to a class of approaches called Software model checking, which aims at analysing full programs for defects (deadlocks, overflows, coverability, etc.). For Java, the most common approach is to instrument an existing virtual machine, running the compiled bytecode, as done by JPF and jMoped for Java and MoonWalker for .NET. This avoids having to implement the semantics of the over 200 bytecodes of Java, some of them being quite complex. Even then, this solution is partial due in particular to I/O and Java's native interface allowing to call arbitrary binary code. It also precludes a complete reuse of existing model checking technologies (SAT-solvers, decision diagrams, etc.), although a few efforts are made (Păsăreanu et al., 2008), since symbolic interpretation of the complete transition relation is impossible.

In contrast, our approach targets *models* expressed in a Java-like syntax. Compared to JPF, VeriJ lacks I/O or any interoperability with existing code, as well as dynamic method resolution since there is no inheritance. However the HFSM model with only eight basic instruction types is enough to formally capture the semantics as an LTS. Moreover, we provide with VeriJ an adapter to use JPF `ChoiceGenerator` with semantics consistent with the `VeriJ.choice` function. Hence, this allows a user familiar with JPF to run additional controls on the VeriJ model. Since JPF, like the other software model checkers, does not have the engine to label the state space in a game structure, we only compare with JPF for the LTS generation. This comparison was also a mean to check the correctness of our transformations with respect to Java semantics.

Table 2 provides this comparison on the automated highway (in addition to $d_{min} = 3$ like in Table 1, we included a case with $n = 10$ and $d_{min} = 4$). Experiments show that JPF consumes less memory but is not as fast as VeriJ. JPF is a tool originally developed by NASA and enhanced over the last decade, but a large quantity of bytecode instructions and space reduction techniques slow it down. This also motivates us to explore more techniques to reduce the memory use.

Table 2 Memory (*MB*) and time (*s*) usage in the automated highway

L	d_{min}	*Number of states*		*Mem*		*Time*	
		VeriJ	*JPF*	*VeriJ*	*JPF*	*VeriJ*	*JPF*
8	3	14,845	67,170	331	76.8	9.1	76.9
10	4	16,489	56,310	321	76.0	9.9	52.6
10	3	32,212	265,506	350	76.7	20.8	685.6

jMoped, a test environment for Java programs, uses pushdown semantics like we do. To be able to perform symbolic execution, the authors had to reimplement the bytecodes semantics. jMoped can deal with multi-threaded unbounded systems. However, due to an incomplete implementation of the Java specification (in particular the Java `random` function is missing), we could not run the tool on our examples.

6.2 Comparison with supervisory control tools

In the domain of controller design, transition systems like FSMs are a natural choice. For instance, Stateflow (http://www.mathworks.fr/products/stateflow) developed by MathWorks provides state charts and flow diagrams. But these tools do not provide controllability check.

Tools applying formal methods for supervisory control usually describe systems with low-level models, for example, FSM (Moor et al., 2010; Miremadi et al., 2008; Behrmann et al., 2007), Petri nets (Zareiee et al., 2012), etc. They share two important common features:

- a graphical interface makes small and simple systems easy to model, but it can be difficult to describe the relationship between complex system elements
- they have efficient techniques to tackle the state space explosion problem.

Among them, the widely used tool Uppaal TIGA (called here Tiga for short, http://www.cs.aau.dk/ adavid/tiga) integrates supervisory control in the expressive Uppaal environment where users can model systems by (possibly timed) automata with additional C-like functions. Tiga supports game automata, with transitions labelled by *controller* or *environment*. It has been used for controller design of industrial systems (David et al., 2012; Jessen et al., 2007) with good performances. There even exists a tool SARTS (Bogholm et al., 2008) that performs a fully automatic translation of real-time Java applications into Uppaal models.

We first compare VeriJ with Tiga on the Nim game. A Tiga model consists of automata templates, text declarations and an extra system definition to compose the templates. The model of the Nim game is depicted in Figure 9 with a single automaton,

where all states are *urgent* (labelled by ∪), because the Nim game does not handle time. Dashed arrows are environment transitions, and plain arrows belong to the controller. For instance, the controller transition from `s2` to `s1` is performed in three steps:

1 choose `r` and `n`

2 check the guard `a[r]>0 && n <=a[r]`

3 take the matches.

To check the controllability, the query is expressed by a logic formula prefixed by the keyword `control`. For this example, we use the formula `control: A[] not (nim.s1 and isEmpty())`, to express that state `s1` is not reached with no matches left, where operator `A` means *for all execution paths* and `[]` stands for *always*.

Figure 9 Modelling Nim game with Uppaal-TIGA (see online version for colours)

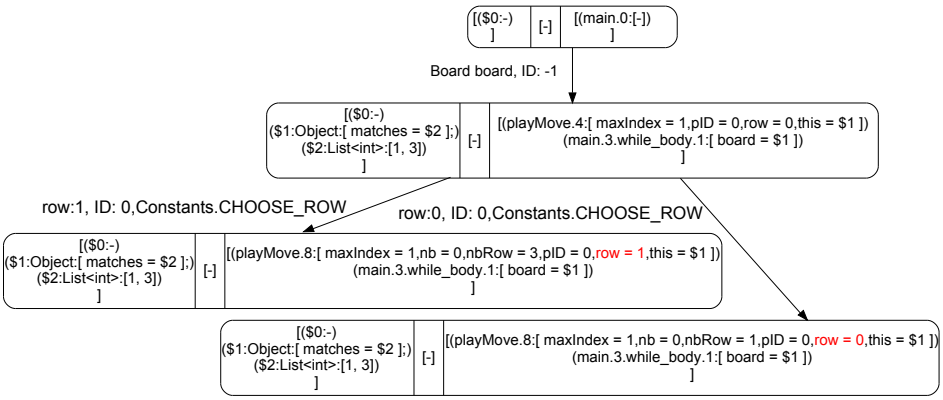


Table 3 Comparison with JPF and Uppaal TIGA on the Nim game

r	*Reachability*						C	*Controllability*			
	Number of states		*Mem (MB)*		*Time (s)*			*Mem (MB)*		*Time (s)*	
	VeriJ	*JPF*	*VeriJ*	*JPF*	*VeriJ*	*JPF*		*VeriJ*	*TIGA*	*VeriJ*	*TIGA*
2	22	9	5	11	0.04	0.3	N	5	2	0.05	0.1
3	204	133	17	17	0.3	0.5	N	17	2	0.4	0.1
4	2.1×10^3	1.5×10^3	46	34	1.1	1.7	Y	47	57	2.0	0.2
5	2.3×10^4	1.8×10^4	132	64	4.9	13.6	N	133	116	12.1	3.6
6	3.1×10^5	2.5×10^5	639	65	61.6	193.3	N	641	-	289.7	-
7	-	3.8×10^6	-	124	-	3,058.5	N	-	-	-	-
8	-	3.9×10^7	-	895	-	31,800	Y	-	-	-	-
9	-	-	-	-	-	-	N	-	-	-	-

Table 3 compares the performances of VeriJ and JPF for reachability, and also compares VeriJ and Tiga for controllability checking (columns on the right), for increasing number of rows (r). For $r = 6$, Tiga runs out of memory, and so do both VeriJ and Tiga when $r > 6$. JPF shows good performance in handling state space (*mem*); while Tiga shows

faster controllability checking. Controllability (C) can be calculated using the solution in Section 2, which permits to check the correctness of the results obtained from the two tools.

Modelling the highway example with Tiga is much more difficult (and not done here) for the following reasons:

- Non-deterministic choice can only be done with a static scope.
- Handling a dynamic number of instances (like a list of cars in the highway) would require some modelling tricks. In particular, the maximum number of instances must be pre-defined.

To sum up, we obtain reasonable performances compared to some of the most mature tools in both areas of Java model checking and controllability.

7 Conclusions

We presented an approach to check controllability of a system described with a high-level programming language. This provides a practical way towards controller design for software engineers, with comfortable modelling environments thanks to Java IDEs.

Users can model systems as VeriJ programs, written in a Java-like syntax. We provide a tool chain to transform such a program into a HFSM, modelling the CFG, and then generate an LTS with pushdown semantics, on which the controllability check can be performed. Several techniques are integrated to increase scalability: garbage collection, abstraction, SC and partial exploration.

Experiments show that our approach is comparable with mature tools from both software model checking and supervisory control. From the perspective of software engineers, VeriJ bridges the gap between a widely-used industrial-strength programming language (Java) and formal methods to deal with the controllability problem.

As future work, we plan to further develop full controller synthesis. We also intend to improve scalability by adapting the techniques from Zhang et al. (2010) to take HFSM as input. Finally, a possible extension would be to support concurrency in VeriJ specifications.

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 *Proc. of the 9th Int. Workshop on Discrete Event Systems (WODES '08)*, pp.346–351, IEEE Press.

Ball, T., Levin, V. and Rajamani, S. (2011) 'A decade of software model checking with SLAM', *Communications of the ACM*, Vol. 54, No. 7, pp.68–76.

Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K. and Lime, D. (2007) 'UPPAAL-TIGA: time for playing games!', in *Proc. of the 19th International Conference on Computer Aided Verification (CAV '07), Lecture Notes in Computer Science*, Vol. 4590, pp.121–125, Springer.

Beyer, D., Henzinger, T.A., Jhala, R. and Majumdar, R. (2007) 'The software model checker blast', *International Journal on Software Tools for Technology Transfer (STTT)*, Vol. 9, No. 5, pp.505–525.

Bogdoll, J., Fioriti, L.F., Hartmanns, A. and Hermanns, H. (2011) 'Partial order methods for statistical model checking and simulation', in *Proc. of Joint 13th IFIP WG 6.1 International Conference on Formal Methods for Open Object-based Distributed Systems and 31st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FMOODS/FORTE), Lecture Notes in Computer Science*, Vol. 6722, pp.59–74, Springer.

Bogholm, T., Kragh-Hansen, H., Olsen, P., Thomsen, B. and Larsen, K. (2008) 'Model-based schedulability analysis of safety critical hard real-time Java programs', in *Proc. of the 6th Int. Workshop on Java Technologies for Real-time and Embedded Systems*, pp.106–114, ACM Press.

Bouton, C.L. (1901) 'Nim, a game with a complete mathematical theory', *The Annals of Mathematics*, Vol. 3, Nos. 1/4, pp.35–39.

Burch, J., Clarke, E., McMillan, K., Dill, D. and Hwang, L. (1992) 'Symbolic model checking: 10^{20} states and beyond', *Information and Computation*, Vol. 98, No. 2, pp.142–170.

Chomsky, N. (1962) *Context-free Grammars and Pushdown Storage*, in Quarterly Progress Report No. 65, pp.187–194, MIT Research Lab. Elect.

Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Zheng, H. et al. (2000) 'Bandera: extracting finite-state models from Java source code', in *Software Engineering, Proceedings of the International Conference on*, pp.439–448, IEEE.

David, A., Grunnet, J., Jessen, J., Larsen, K. and Rasmussen, J. (2012) 'Application of model-checking technology to controller synthesis', in *Formal Methods for Components and Objects*, Vol. 6957 of *LNCS*, pp.336–351, Springer.

Flanagan, C. and Qadeer, S. (2002) 'Predicate abstraction for software verification', *SIGPLAN Notices*, Vol. 37, No. 1, pp.191–202.

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, Lecture Notes in Computer Science*, p.152, Springer.

Holzmann, G.J. (1997) 'State compression in SPIN: recursive indexing and compression training runs', in *Proc. of the 3nd International SPIN Workshop*.

Ivančić, F., Yang, Z., Ganai, M., Gupta, A. and Ashar, P. (2008) 'Efficient SAT-based bounded model checking for software verification', *Theoretical Computer Science*, Vol. 404, No. 3, pp.256–274.

Jessen, J., Rasmussen, J., Larsen, K. and David, A. (2007) 'Guided controller synthesis for climate controller using UPPAAL TIGA', in *Proc. of the 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS '07), Lecture Notes in Computer Science*, Vol. 4763, pp.227–240, Springer.

Laarman, A., Pol, J. and Weber, M. (2011) 'Parallel recursive state compression for free', in *Proc. of the 18th International SPIN Workshop, Lecture Notes in Computer Science*, Vol. 6823, pp.38–56, Springer.

Lime, D., Roux, O., Seidner, C. and Traonouez, L. (2009) 'Romeo: a parametric model-checker for Petri nets with stopwatches', in *Tools and Algorithms for the Construction and Analysis of Systems, 15th Int. Conference, TACAS*, Vol. 5505 of *LNCS*, pp.54–57, Springer.

Miremadi, S., Akesson, K., Fabian, M., Vahidi, A. and Lennartson, B. (2008) 'Solving two supervisory control benchmark problems using supremica', in *the 9th Int. Workshop on Discrete Event Systems (WODES '08)*, pp.131–136, IEEE.

Moor, T., Schmidt, K. and Perk, S. (2010) 'Applied supervisory control for a flexible manufacturing system', in *11th Int. Workshop on Discrete Event Systems (WODES '10)*, pp.253–258, IFAC/Elsevier.

Oettinger, A.G. (1961) 'Automatic syntactic analysis and the pushdown store', in *Structure of Language and its Mathematical Aspects*, Vol. 12 of *Symposia on Applied Mathematics*, pp.104–129, American Mathematical Society.

Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S. and Pape, M. (2008) 'Combining unit-level symbolic execution and system-level concrete execution for testing NASA software', in *Proc. of the Int. Symposium on Software Testing and Analysis, ISSTA*, pp.15–26, ACM Press.

Ramadge, P.J. and Wonham, W.M. (1987) 'Supervisory control of a class of discrete event processes', *SIAM Journal on Control and Optimization*, Vol. 25, No. 1, pp.206–230.

Sistla, A. and Godefroid, P. (2004) 'Symmetry and reduced symmetry in model checking', *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 26, No. 4, pp.702–734.

Zareiee, M., Dideban, A., Orouji, A. and Alla, H. (2012) 'A simple Petri net controller in discrete event systems', in *14th IFAC Symposium on Information Control Problems in Manufacturing*, Vol. 14, pp.188–193.

Zhang, Y., Bérard, B., Kordon, F. and Thierry-Mieg, Y. (2010) 'Automated controllability and synthesis with hierarchical set decision diagrams', in *Proc. of the 11th Int. Workshop on Discrete Event Systems (WODES)*, pp.291–296, IFAC/Elsevier.

Zhang, Y., Bérard, B., Hillah, L.M., Kordon, F. and Thierry-Mieg, Y. (2011) 'Modeling complex systems with VeriJ', in *Proc. of the 5th Int. Workshop on Verification and Evaluation of Computer and Communication System (VECOS)*, pp.34–45, British Computer Society.

Zhang, Y. (2010) 'Modeling automated highway systems with VeriJ', in *Modelling and Verifying Parallel Processes (MOVEP '10)*, pp.138–143.