# A High-Level Model Checking Language with Compile-time Pruning of Local Variables

Giovanni Pardini and Paolo Milazzo

Dipartimento di Informatica, Università di Pisa
Largo B. Pontecorvo, 3, 56127 Pisa
{pardinig,milazzo}@di.unipi.it

**Abstract.** Among Model Checking tools, the behaviour of a system is often formalized as a transition system with atomic propositions associated with states (*Kripke structure*). In current modelling languages, transitions are usually specified as updates of the system's variables to be performed when certain conditions are satisfied. However, such a low-level representation makes the description of complex transformations difficult, in particular in the presence of structured data.
We present a high-level language with imperative semantics for modelling finite-state systems. The language features are selected with the aim of enabling the translation of models into compact transition systems, amenable to efficient verification via Model Checking. To this end, we have developed a compiler of our high-level language into the modelling language of the PRISM probabilistic model checker.
One of the main characteristics of the language is that it makes a very different treatment of global and local variables. It is assumed that global variables are actually the variables that describe the state of the modelled system, whereas local variables are only used to ease the specification of the system's internal mechanisms. In this paper we describe the procedure for the pruning of local variables that is executed at compile time.

## 1 Introduction

Expressing the model of a system in order to analyze it by means of model checking can require time. Modelling in itself is often a challenging task, since it requires (i) understanding the mechanisms that govern the dynamics of the system, (ii) performing suitable abstractions that allow such mechanisms to be expressed in a concise way, and (iii) constructing an unambiguous representation of the system mechanisms at the chosen abstraction level, by exploiting a notation that usually depends on the analysis method one wants to apply.

In the case of model analysis by means of model checking, the system dynamics has usually to be formalized as a transition system. Often, such a transition system has the form of a *Kripke structure*, namely it has atomic propositions associated with states, which are used as the basis for the specification of properties to be verified. The way in which a transition system is expressed in the input languages of model checking tools can vary significantly from tool to tool.

In many cases the input language allows transitions to be specified as updates of the system's variables to be performed when certain conditions are satisfied (as if-then clauses or rewrite rules). However, if-then clauses and rewrite rules are often too low level as a modelling paradigm to express systems' mechanisms in a natural way. Such a low level nature often requires modellers to formulate complex expressions as transition conditions, or to introduce in the model unnecessary variables and transitions just to represent intermediate values for the computation of the next state reached by a transition. This situation is particularly frequent in the case of languages that offer no or few data structures.

The aim of this paper is to propose a new language, called Objective/MC, for the specification of system models to be analysed by means of model checking. We plan to include in Objective/MC rich data structures, object-oriented features and concurrent processes. In this paper, however, we start with the definition of a core version of the language with the main imperative constructs and a basic notion of array and of object.

One of the main characteristics of the language is that it makes a very different treatment of global and local variables. It is assumed that global variables are actually the variables that describe the state of the modelled system, whereas local variables are only used to ease the specification of the system's internal mechanisms. Hence, in this paper we focus on the problem of pruning local variables and generating transitions that correspond to updates of global variables.

The language is developed with the aim of allowing translation of Objective/MC models into the input language of an existing model checking tool. Since in the long term we plan to allow probabilistic and stochastic systems to be dealt with, we choose PRISM as the target model checking tool in this paper. We also include in the language an operator to perform non-deterministic choices that are translated into probabilistic choices in PRISM.

An implementation of the translator from Objective/MC into the modelling language of the PRISM model checker is available [1]. The translator transforms the model in order to make all local variables disappear. Hence, the generated transition system will consist of transitions that correspond to the updates of only the global variables used in the model.

*Related works* Among model checkers with an input language essentially based on if-then clauses or rewrite rules we mention NuSMV [2], PRISM [7] and UPPAAL [8]. In these cases the state of system is merely represented as a set of variables or by simple data structures.

Maude [4] offers a declarative input language based on rewrite rules with the possibility of using complex structures such as terms and objects. The Maude language is presently richer than Objective/MC in terms of functionalities. Differently, Objective/MC is an imperative language and offers compile-time pruning of local variables to combine the ease of specification with the generation of a compact transition system. Moreover, Objective/MC is already designed with the aim of extending it with rich data structures and object oriented features.

Other tools have rich modelling languages, the features of which are often related with the specific kind of systems addressed. This is for instance the case

**Declarations**

| | | | |
|---|---|---|---|
| $Model$ | $::=$ | $\overline{D}\ \overline{G}\ \overline{M}\ \mathbf{run}\ \{\overline{Com}\}$ | *(model)* |
| $D$ | $::=$ | $\mathbf{class}\ c\ \{\overline{F}\}$ | *(class)* |
| $M$ | $::=$ | $\mathbf{void}\ w(\overline{T_\mathrm{v}}\ \overline{I})\ \{\overline{Com}\}$ | *(proc.)* |
| $F, G$ | $::=$ | $T\ I$ | *(field/global var)* |

**Commands**

| | | | |
|---|---|---|---|
| $Com$ | $::=$ | $T_\mathrm{v}\ I\ =\ E$ | *(local var decl)* |
| | $\mid$ | $Loc = E$ | *(assignment)* |
| | $\mid$ | $\{\overline{Com}\}$ | *(block)* |
| | $\mid$ | $\mathbf{if}\ (E)\ \{\overline{Com}\}\ \mathbf{else}\ \{\overline{Com}\}$ | *(if)* |
| | $\mid$ | $\mathbf{while}\ (E)\ \{\overline{Com}\}$ | *(while)* |
| | $\mid$ | $\mathbf{times}\ (E)\ \{\overline{Com}\}$ | *(times)* |
| | $\mid$ | $w(\overline{E})$ | *(procedure call)* |
| | $\mid$ | $\mathbf{yield}$ | *(yield)* |

**Types**

| | | | |
|---|---|---|---|
| $T_\mathrm{v}$ | $::=$ | $\mathbf{int}(n\mathinner{..}m)\ \mid\ \mathbf{bool}$ | *(basic)* |
| $T$ | $::=$ | $T_\mathrm{v}\ \mid\ c\ \mid\ T[n]$ | *(compound)* |

**Expressions**

| | | | |
|---|---|---|---|
| $Loc$ | $::=$ | $I$ | *(identifier)* |
| | $\mid$ | $Loc[E]$ | *(array)* |
| | $\mid$ | $Loc.f$ | *(field)* |
| $E$ | $::=$ | $v$ | *(literal value)* |
| | $\mid$ | $Loc$ | *(location)* |
| | $\mid$ | $E\ op_\mathrm{b}\ E$ | *(binary op.)* |
| | $\mid$ | $op_\mathrm{u}\ E$ | *(unary op.)* |
| | $\mid$ | $E\ ?\ E : E$ | *(conditional)* |
| | $\mid$ | $E\ \mathbf{as}\ \mathbf{int}(n\mathinner{..}m)$ | *(ascription)* |
| | $\mid$ | $\mathbf{select}(n, m)$ | *(select)* |

Fig. 1: Syntax of Objective/MC, where $v$ denotes a value in $\mathbb{Z} \cup \{\mathbf{true}, \mathbf{false}\}$, $op_\mathrm{b} \in \{+, -, *, /, <, \leq, >, \geq, ==, !=, \&\&, \|\}$, $op_\mathrm{u} \in \{+, -, \neg\}$.

of Software Model Checkers (such as SPIN [6], CBMC [3] or Java Pathfinder [5]) that are based on rich languages for the specification of concurrent processes, or can even executed directly on C or Java source code. In particular, as regards SPIN (and its input language Promela), it includes high level imperative programming constructs and concurrency features. The same holds for Rebeca (Reactive Objects Language) [9], an actor-based language for modelling and verification of reactive systems. With respect to these languages, Objective/MC offers the novelty of the compile-time pruning of local variables.

We remark that the long-term aim of our work is to propose a new high-level modelling language with imperative semantics. The definition of the local variable pruning methodology is a step in this direction. For this reason, we have defined the methodology on a new language rather than on an existing language.

## 2 The Core Objective/MC Language

In this section, we present a *core* version of the Objective/MC language, which allows us to focus on the foundational aspects of the translation of models into transition systems, in particular as regards the handling of global/local variables. The core language provides only a limited support for object-oriented features; an extensive treatment of those aspects will be the object of a future paper.

The syntax of the Objective/MC language is shown in Figure 1. For conciseness, we denote a sequence of symbols (of either a syntactical category or terminal symbols) by using an overbar. The concrete syntax follows C-like conventions, such as using semicolons as command terminators.

A *Model* is composed of sequences of *class declarations* $\overline{D}$, *global variables declarations* $\overline{G}$, and *procedure declarations* $\overline{M}$ (akin to static methods). In a procedure declaration, the notation $\overline{T_v}\ \overline{I}$ denotes a sequence of pairs $T_v^1\, I^1, \ldots, T_v^k\, I^k$.

The language provides two basic data types: *bounded integers*, declared as $\mathbf{int}(n \mathinner{..} m)$ with $n \leq m$, which can hold any integer value $x$ in the range $[n, m]$[1], and *booleans*, declared as **bool**, holding values $\{\mathbf{true}, \mathbf{false}\}$. There are also two compound types: fixed-length *arrays*, and *classes*. Each definition of a class introduces a type (different from any other type) which is identified by the name of the class itself. A class contains a number of *fields* $\overline{F}$ of any type. However, only class types which have been declared previously may be referenced, thus ruling out any recursive type.

Global variables are statically allocated, and they are meant to constitute the global state of the system which is updated by the assignment statements occurring in the program model. There are no restriction on the types of the global variables; that is, any valid type according to the grammar, and referring only to classes already declared, may be used. In particular, this allows objects and arrays to be defined as global variables. On the other hand, objects/arrays can be neither defined as local variables nor dynamically allocated. As regards the declaration of procedures $\overline{M}$, they may take any number of parameters, and their types are restricted to the basic types.

The main entry point of the program model is the **run** command block. Commands are executed sequentially, and the canonical control flow commands (*if*, *while*, procedure calls) are provided by the language. Actually, there are two different cycle construct: a standard **while** $(E)$ {...} command which executes its body as long as the guard $E$ evaluates to *true*, and a **times** $(E)$ {...} command which instead executes the body for a fixed number of iterations, obtained by evaluating the integral-type expression $E$. Recursion is not allowed.

Local temporary variables having basic type **int**/**bool** may be declared and used, and they must be initialized at the same time of their declarations. The syntax of *expressions* is quite standard, where the syntactical category *Loc* for *locations* is used to access: (i) local/global variables from their *identifier* $I$, (ii) elements of *arrays* as $Loc[E]$, or (iii) *object fields* as $Loc.f$. A *non-deterministic* construct $\mathbf{select}(n, m)$, with $n, m \in \mathbb{Z}$, is also provided, which "generates" an integral value in the range $[n, m]$ every time the expression is evaluated.

An assignment of the form $Loc = E$ may refer to either a global or local variable, where $E$ is an expression which evaluates to a basic type, and *Loc* must refer to a variable of a *compatible* type. Formally, either both *Loc* and $E$ are expressions of type **bool**, or the type of the right hand side expression is an integral type $\mathbf{int}(n_2 \mathinner{..} m_2)$ with range fully contained in the range of the left hand side variable $\mathbf{int}(n_1 \mathinner{..} m_1)$, namely $[n_2, m_2] \subseteq [n_1, m_1]$. In order to fulfil this requirement, the *ascription* construct $E$ **as** $\mathbf{int}(n \mathinner{..} m)$ may be used to force an expression to be considered in the type system as a different (stricter) integral type than the one which can be derived from the subexpressions of $E$.[2]

[1] The notation $[n, m]$ indicates the integral range of values $\{x \mid n \leq x \leq m\} \subset \mathbb{Z}$.

[2] The formal definition of the type system is omitted due to lack of space.

**Semantics** The semantics of a model $M$ is a transition system, whose *states* are characterized by the all and only global variables defined in the program model (except for a hidden global "program counter" variable needed for the translation of sequential commands). As a consequence, assignments to global variables give rise to transitions in the resulting transition system. A special command **yield** is used to generate a transition for which no declared global variables are modified, and actually only the hidden program counter is updated. Since the Objective/MC language is geared towards the model checking of the resulting transition systems, in defining the semantics we strictly adhere to the following principle:

*No transitions may be implicitly introduced by the compilation process beyond those transitions which correspond to updates to global variables.*

The motivation for the above principle is that we want to avoid enlarging the state space implicitly. Nevertheless, the modeller may always introduce global variables if necessary to describe an intended behaviour, which *explicitly* enlarge the state space. In the current definition of the language, each *single* assignment to a global variable causes a transition in the resulting transition system.

The language allows *local variables* to be defined and used according to their intuitive imperative semantics. However, rather than storing them in the global state, the compiler instantiates each access to a local variable with an equivalent expression containing only literal values and references to global variables. For this reason, some limitations on their usage apply.

**Constraints** Due to the semantics of Objective/MC, there are three constraints on the usage of the language, exemplified by the following model fragments.

```
1 int(0..5) x = g + 1;
2 if (x > 0)
3     g = 0; // global assm.
4 else
5     q = 0; // global assm.
6 int(0..6) y = 0;
7 y = x + 1;      // error
```

(a)

```
1 int (0..5) x = 0;
2 while (g > 0) {
3     if (q == x)
4         x = g;   // error
5     q = g; // global assm.
6 }
7 int(0..6) y = x;
```

(b)

```
1 while (g > 0) {
2     if (q < g) {
3         g = 0; // global assm.
4     } else {
5         // empty
6     }
7 }   // error in 'while' loop
```

(c)

(a) *A local variable cannot be read after the global variables it depends upon may have been modified.* In the example, at line 1, the local variable `x` is initialized with the global variable `g`, and it is then *accessed* (i.e. *read*) at line 7. However, since local variables are handled as unevaluated expressions by the compiler, at line 7 its definition may have been invalidated due to the assignment to the dependant global variable `g` at line 3. A possible solution is to promote variable `x` into a global variable, thus allowing the previous value of `g + 1` to be retained across transitions.

(b) *A local variable declared outside of a* while *loop cannot be reassigned in the loop body.* Since the loop may be executed an unbounded number of times, it is generally impossible to know the value of a local variable which may

```
 1 int(0..10)[5] P;          // array of five processors
 2 int(0..10) nextjob;       // duration of the next job
 3 void execute_step() {     // decreases the load of each processor by one unit
 4     int(0..4) i = 0;
 5     times (5) {
 6         if (P[i] > 0) { P[i] = P[i] as int(1..10) - 1; }
 7         i = (i < 4 ? i + 1 : i) as int(0..4); }
 8 }
 9 run {
10     while (true) {
11         execute_step();
12         if (nextjob == 0) nextjob = select(0,3);
13         int(0..4) min_idx = 0;
14         int(0..4) i = 1;
15         times (4) {
16             if (P[i] < P[min_idx]) min_idx = i;
17             i = (i < 4 ? i + 1 : i) as int(0..4); }
18         if (P[min_idx] + nextjob <= 10) {
19             P[min_idx] = (P[min_idx] + nextjob) as int(0..10);
20             nextjob = 0;
21         } else yield;
22     }
23 }
```

Fig. 2: An Objective/MC model of a job processor scheduler.

be reassigned inside the loop body. In the example above, the local variable `x` may be assigned at line 4 hence (analogously to the previous case) when it is read at lines 3 and 7, its value cannot be definitely known. Note that *times* loops do not have this limitation.

(c) *Each execution path inside the body of a* while *loop must contain at least one global variable assignment or a* yield*.* This constraint is useful to simplify the translation, since it frees the compiler from the need to identify infinite loops at compile-time, as in the example, for which an infinite loop occurs if $0 < g \leq q$. A possible solution is to insert a **`yield`** command at line 5, in such a way that a transition is generated for each branch of the **`if`**.

Constraints (a) and (b) avoid situations in which one or more transitions in the semantics of the model make it impossible to reconstruct the value of the local variable from current values of the global variables. This could be due either, in case (a), to a change in a global variable upon which the local variable depends, or, in case (b), to the impossibility of knowing the function that allows the local variable to be computed from the global ones. Hence, these language constraints reflect potential constraints arising from the model itself, namely situations in which information needs to be included in the states of the model.

*Example 1.* Fig. 2 contains a model of a *job scheduler* for a fixed number of processors, which may execute jobs of different durations. A processor is characterized by its *load*, modelled as an integer in the range $[0, 10]$. At line 1, an array of 5 processors is declared as a global variable `P`. Each element of the array is initialized by default at 0. At line 2, a global variable `nextjob` is declared, representing the duration of the next job to be scheduled (initially 0).

At line 3, a procedure with no parameters is declared. Its purpose is to decrease the load of each processor by one unit each time it is invoked, in order to simulate the progress of the system. A temporary local variable `i` is used as the index in the array of processors, and as such it will not appear in the translated model. At each iteration of the **`times`** cycle, the current processor load `P[i]`, if it is not already nil, is decreased by the assignment at line 6.

The system executes an infinite loop where, at each iteration: (i) procedure `execute_step()` is called, (ii) if there are no jobs waiting (`nextjob` $= 0$), the duration of the next job is selected non-deterministically in the range $[0, 3]$ (line 12), (iii) the index of the least-loaded processor (`min_idx`) is computed (lines 13–17), and finally (iv) the new job is tentatively assigned to processor `P[min_idx]` if it can accommodate it and, in this case, `nextjob` is reset (lines 18–21).

The **`yield`** command at line 21 is necessary in order to satisfy constraint (c), since it might be the case that no assignments to global variables are executed in an iteration of the **`while`** loop.

## 3 Control Flow Graph

The **run** block is translated into a *Control Flow Graph* (CFG), which allows us to abstract from the syntactical representation of the model, and simplify the definition of the subsequent transformations. Let $\mathit{\Pi}_{\text{in}}$ and $\mathit{\Pi}_{\text{out}}$ be countable disjoint sets of *input* and *output endpoints*, respectively. A CFG $G$ is a pair $(\mathrm{N}(G), \mathrm{E}(G))$ composed of a set of nodes $\mathrm{N}(G)$ of the form $[\mathbf{T}]^{\pi_0}_{\pi_1,\ldots,\pi_k}$, with $\pi_0 \in \mathit{\Pi}_{\text{in}}$ and $\{\pi_1, \ldots, \pi_k\} \subset \mathit{\Pi}_{\text{out}}$, and a set of edges $\mathrm{E}(G) \subset \mathit{\Pi}_{\text{out}} \times \mathit{\Pi}_{\text{in}}$ such that $\forall(\alpha, \beta_1), (\alpha, \beta_2) \in \mathrm{E}(G).\ \beta_1 = \beta_2$. Assuming $\mathrm{input}([\mathbf{T}]^{\pi_0}_{\pi_1,\ldots,\pi_k}) = \pi_0$ and $\mathrm{outputs}([\mathbf{T}]^{\pi_0}_{\pi_1,\ldots,\pi_k}) = \{\pi_1, \ldots, \pi_k\}$, we define, for any $N \subseteq \mathrm{N}(G)$: $\mathrm{inputs}(N) = \bigcup_{\mathbf{x}\in N}\{\mathrm{input}(\mathbf{x})\}$; $\mathrm{outputs}(N) = \bigcup_{\mathbf{x}\in N} \mathrm{outputs}(\mathbf{x})$; $\mathrm{endpoints}(N) = \mathrm{inputs}(N) \cup \mathrm{outputs}(N)$. For conciseness, we also define $\mathrm{outputs}(\pi_0) = \{\pi_1, \ldots, \pi_k\}$.

Let $G$ be a graph. Given $\pi \in \mathrm{inputs}(\mathrm{N}(G))$, its *predecessors* are defined as $\mathrm{pred}_G(\pi) = \{\alpha \mid (\alpha, \pi) \in \mathrm{E}(G)\}$, while, given $\pi \in \mathrm{outputs}(\mathrm{N}(G))$, its *successors* are defined as $\mathrm{succ}_G(\pi) = \{\alpha \mid (\pi, \alpha) \in \mathrm{E}(G)\}$. Note that $|\mathrm{succ}_G(\pi)| \leq 1$.

### 3.1 Construction of the Control Flow Graph

The types of nodes defined for the Control Flow Graph are the following.

$$[\mathbf{let}\ I = E]^{\pi_0}_{\pi_1} \quad [\ell \Leftarrow E]^{\pi_0}_{\pi_1} \quad [\mathbf{yield}]^{\pi_0}_{\pi_1} \quad [\mathbf{if}\ E]^{\pi_0}_{\pi_1,\pi_2} \quad [\mathbf{select}_k]^{\pi_0}_{\pi_1,\ldots,\pi_k}$$
$$[\mathbf{begin}]^{\pi_0}_{\pi_1} \quad [\mathbf{end}]^{\pi_0} \quad [\mathbf{skip}]^{\pi_0}_{\pi_1}$$

The first two nodes represent assignments to either a *local* variable $I$ or a global variable $\ell \in \mathit{Loc}$, respectively, while the third directly corresponds to the *yield* command. The *if* node is used to translate both *if* and *while* commands, where the output endpoints $\pi_1, \pi_2$ denote the branches to follow when the condition

$$\frac{\ell \neq I \text{ or } \ell \in \mathit{Globals}}{[\![\ell = E]\!] \mapsto [\ell \Leftarrow E]^{\pi_0}_{\pi_1}, \pi_0, \{\pi_1\}} \qquad \frac{I \notin \mathit{Globals}}{[\![I = E]\!] \mapsto [\textbf{let } I = E]^{\pi_0}_{\pi_1}, \pi_0, \{\pi_1\}}$$

$$\frac{}{[\![T_{\mathrm{v}}\, I = E]\!] \mapsto [\textbf{let } I = E]^{\pi_0}_{\pi_1}, \pi_0, \{\pi_1\}} \qquad \frac{}{[\![\textbf{yield}]\!] \mapsto [\textbf{yield}]^{\pi_0}_{\pi_1}, \pi_0, \{\pi_1\}}$$

$$\frac{[\![\overline{C_1}]\!] \mapsto G_1, \mathit{in}_1, \mathit{out}_1 \qquad [\![\overline{C_2}]\!] \mapsto G_2, \mathit{in}_2, \mathit{out}_2 \qquad \mathit{out}' = \mathit{out}_1 \cup \mathit{out}_2}{[\![\textbf{if } (E)\ \{\overline{C_1}\} \textbf{ else } \{\overline{C_2}\}]\!] \mapsto G_1 + G_2 + [\textbf{if } E]^{\pi_0}_{\pi_1,\pi_2} + (\pi_1, \mathit{in}_1) + (\pi_2, \mathit{in}_2), \pi_0, \mathit{out}'}$$

$$\frac{[\![\overline{\mathit{Com}}]\!] \mapsto G, \mathit{in}, \mathit{out}}{[\![\textbf{while } (E)\ \{\overline{\mathit{Com}}\}]\!] \mapsto G + [\textbf{if } E]^{\pi_0}_{\pi_1,\pi_2} + (\pi_1, \mathit{in}) + (\mathit{out} \times \{\pi_0\}), \pi_0, \{\pi_2\}}$$

$$\frac{\operatorname{type}(E) = \textbf{int}(n_0..m) \qquad n = \max(0, n_0) \qquad \forall i \in [1, m].\ [\![\overline{\mathit{Com}}]\!] \mapsto G_i, \mathit{in}_i, \mathit{out}_i}{\begin{array}{l}[\![\textbf{times } (E)\ \{\overline{\mathit{Com}}\}]\!] \mapsto \sum_{i=1}^{m} G_i + \sum_{i=2}^{m} (\mathit{out}_i \times \{\mathit{in}_{i-1}\}) + \sum_{i=n}^{m} [\textbf{if } E == i]^{\alpha_i}_{\beta_i,\gamma_i} \\ \qquad\qquad + \sum_{i=n}^{m} (\beta_i, \mathit{in}_i) + \sum_{i=n+1}^{m} (\gamma_i, \alpha_{i-1}),\ \alpha_m,\ \mathit{out}_1 \cup \{\gamma_n\}\end{array}}$$

$$\frac{\operatorname{decl}(w) = w(T_1\, I_1, \ldots, T_k\, I_k)\ \{\overline{\mathit{Com}}\} \text{ with } k > 0 \qquad [\![\overline{\mathit{Com}}]\!] \mapsto G, \mathit{in}, \mathit{out}}{[\![w(E_1, \ldots, E_k)]\!] = G + \sum_{i=1}^{k} [\textbf{let } I_i = E_i]^{\alpha_i}_{\beta_i} + \sum_{i=1}^{k-1} (\beta_i, \alpha_{i+1}) + (\beta_k, \mathit{in}), \alpha_1, \mathit{out}}$$

$$\frac{\operatorname{decl}(w) = w()\ \{\overline{\mathit{Com}}\}}{[\![w()]\!] = [\![\overline{\mathit{Com}}]\!]} \qquad \frac{}{[\![\{\overline{\mathit{Com}}\}]\!] = [\![\overline{\mathit{Com}}]\!]} \qquad \frac{|\overline{\mathit{Com}}| = 0}{[\![\overline{\mathit{Com}}]\!] \mapsto [\textbf{skip}]^{\pi_0}_{\pi_1}, \pi_0, \{\pi_1\}}$$

$$\frac{[\![\mathit{Com}]\!] \mapsto G_1, \mathit{in}_1, \mathit{out}_1 \qquad [\![\overline{\mathit{Com}}]\!] \mapsto G_2, \mathit{in}_2, \mathit{out}_2 \qquad |\overline{\mathit{Com}}| > 0}{[\![\mathit{Com}\ \overline{\mathit{Com}}]\!] \mapsto G_1 + G_2 + (\mathit{out}_1 \times \{\mathit{in}_2\}), \mathit{in}_1, \mathit{out}_2}$$

Fig. 3: Construction rules for the Control Flow Graph.

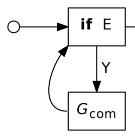


Fig. 4: *While* loop.

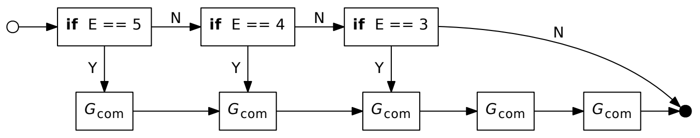


Fig. 5: *Times* loop, when type$(E) = \textbf{int}(3 .. 5)$.

$E$ is either satisfied or not, respectively. The *select* node is parameterized by the number of choices $k \geq 1$. There is also a *begin* node, an *end* node (without output endpoints), and a *skip* node for empty command blocks.

Given two graphs $G_1, G_2$, we define a *union* operation as $G_1 + G_2 = (\mathrm{N}(G_1) \cup \mathrm{N}(G_2), \mathrm{E}(G_1) \cup \mathrm{E}(G_2))$, provided that $\operatorname{endpoints}(\mathrm{N}(G_1)) \cap \operatorname{endpoints}(\mathrm{N}(G_2)) = \emptyset$. Moreover, we define the *addition* of a set of edges $E$ to a graph $G$ as $G + E = (\mathrm{N}(G), \mathrm{E}(G) \cup E)$, provided that $E \subseteq \operatorname{outputs}(\mathrm{N}(G)) \times \operatorname{inputs}(\mathrm{N}(G))$. Given $G_1, G_2$ such that $\mathrm{N}(G_2) \subseteq \mathrm{N}(G_1)$ and $\mathrm{E}(G_2) \subseteq \mathrm{E}(G_1)$, we define a *difference* operation as $G_1 - G_2 = (N', E')$ where $N' = \mathrm{N}(G_1) \setminus \mathrm{N}(G_2)$, and $E' = (\mathrm{E}(G_1) \setminus \mathrm{E}(G_2)) \cap (\operatorname{outputs}(N') \times \operatorname{inputs}(N'))$.

The translation of (a sequence of) commands $\overline{\mathit{Com}}$ is obtained through the relation $[\![\overline{\mathit{Com}}]\!] \mapsto \langle G, \mathit{in}, \mathit{out}\rangle$, where $\mathit{in} \in \operatorname{inputs}(\mathrm{N}(G))$ and $\mathit{out} \subseteq \operatorname{outputs}(\mathrm{N}(G))$ are the designated *input endpoint* and *output endpoints* of the whole graph, respectively. Relation $[\![\cdot]\!]$ is the smallest relation satisfying the rules shown in Fig. 3 (brackets $\langle\ \rangle$ are omitted for clarity) where, for conciseness, a single node $[\mathbf{T}]^{\pi_0}_{\pi_1,\ldots,\pi_k}$ also denotes the graph composed of only such a node. Moreover, we

```
int(-50..50) Q = 0;
run {
   while (Q != -50 && Q != 50) {
      int(0..1) x = select(0,1);
      if (x == 0)
         Q = (Q - 1) as int(-50..50);
      else
         Q = (Q + 1) as int(-50..50);
   }
}
```

Fig. 6: A *random walk* model.

assume local variable names (including parameters) to be unique, and distinct from the global ones (denoted *Globals*); decl$(w)$ to denote the declaration of a procedure $w$; and type$(E)$ the type of $E$ as inferred by the type system. The definition is quite straightforward. We just point out the different translation of *while* and *times* loops, depicted in Fig. 4, 5 (the designated *input* and *output* endpoints are depicted with symbols $\circ$ and $\bullet$, respectively). In the *times* case, the type of $E$ determines the number of copies of the translated loop body $G_{\text{com}}$ (obtained from $[\![\overline{Com}]\!]$) in the CFG. Finally, the resulting CFG is obtained by connecting a *begin* node to the designated input endpoint *in*, and by connecting the *out* endpoints to an *end* node. Given a graph $G$, we denote the input endpoint of the *begin* and *end* nodes by begin$(G)$ and end$(G)$, respectively.

*Example 2.* Fig. 6 shows an implementation of a one-dimensional *random walk*.[3] The position is modelled as a global variable `Q`, initialized at 0, which can vary in the range $[-50, 50]$. The process goes on until the limits of the allowed range for the position are reached. At each iteration of the *while* loop, the position is either decreased or increased by 1, in a non-deterministic way, according to the choice performed by the `select(0,1)` operation. Its CFG is shown in Fig. 7a.

## 3.2 Transformations

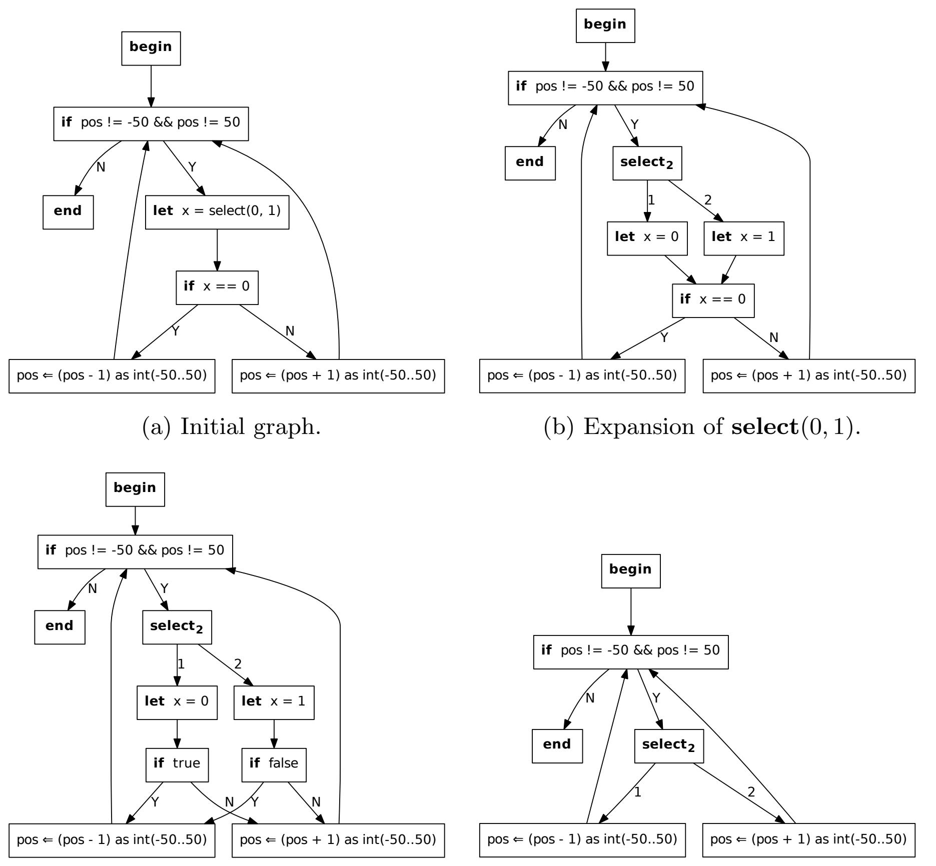
The first transformation consists in the elimination of the *skip* nodes; that is, for all nodes $\mathbf{x} = [\mathbf{skip}]^{\pi_0}_{\pi_1} \in \mathrm{N}(G)$, we apply $G \rightsquigarrow G - \mathbf{x} + (\mathrm{pred}_G(\pi_0) \times \mathrm{succ}_G(\pi_1))$.

The second transformation concerns the *expansion* of *select* operators, and consists in replacing each node containing one or more $\mathbf{select}(n, m)$ operators with an explicit $[\mathbf{select}_k]$ node, with $k = m - n + 1$. We introduce the function alt : $E \to \mathscr{P}(E)$ which expands an expression $E$ into a set of expressions, one for each different combination of values which can be generated by the all *select* operators appearing in $E$. We define $\mathrm{alt}(v) = \{v\}$, $\mathrm{alt}(I) = \{I\}$, $\mathrm{alt}(\mathbf{select}(n, m)) = \{n, \ldots, m\}$ and, for any other expression operator $\varphi \neq \mathbf{select}$, $\mathrm{alt}(\varphi(E_1, \ldots, E_n)) = \bigcup\{\varphi(E'_1, \ldots, E'_n) \mid E'_i \in \mathrm{alt}(E_i),\ i = [1, n]\}$. The actual transformations applied are shown in Fig. 8.

**Normalization** The next transformations involve *let* nodes for local variables. The aim is to obtain an equivalent *normalized* CFG, namely such that for any node in which a local variable $I$ is *read*, the definition of $I$ is univocally determined. To this purpose, two different transformations are needed, which are based on the duplication of parts of the CFG.

We define, for each $\pi \in \mathrm{endpoints}(\mathrm{N}(G))$, the relation $\mathrm{RD}_\pi \subseteq I \times \mathrm{inputs}(\mathrm{N}(G))$ such that $(I, \alpha) \in \mathrm{RD}_\pi$ implies that there exists a path from $\pi$ to an ancestor

[3] In this example, we have assumed that global variable `Q` can be initialized inline.


begin
if pos != -50 && pos != 50
N
Y
end
let x = select(0, 1)
if x == 0
pos ⇐ (pos - 1) as int(-50..50)
pos ⇐ (pos + 1) as int(-50..50)
select₂
1
2
let x = 0
let x = 1
if true
if false

(a) Initial graph. (b) Expansion of **select**$(0, 1)$.

(c) Normalization step. (d) Final graph.

Fig. 7: Various transformations of the CFG in the *random walk* example.

$\alpha$ of the form $[\textbf{let}\ I = E]^{\alpha}_{\beta}$, with no other intervening definitions of $I$ along the path. We write $\mathrm{RD}_{\pi}(I) = \{\alpha \mid (I, \alpha) \in \mathrm{RD}_{\pi}\}$. Formally, RD is defined by the following equations derived from the nodes of the CFG:

$$\begin{aligned}
[\textbf{begin}]^{\pi_0}_{\pi_1} &\longmapsto \mathrm{RD}_{\pi_1} = \mathrm{RD}_{\pi_0} = \emptyset \\
[\textbf{let}\ I = E]^{\pi_0}_{\pi_1} &\longmapsto \mathrm{RD}_{\pi_1} = (\mathrm{RD}_{\pi_0} \setminus \{(I, \alpha) \in \mathrm{RD}_{\pi_0}\}) \cup \{(I, \pi_0)\} \\
[\ell \Leftarrow E]^{\pi_0}_{\pi_1},\ [\textbf{yield}]^{\pi_0}_{\pi_1} &\longmapsto \mathrm{RD}_{\pi_1} = \mathrm{RD}_{\pi_0} \\
[\textbf{if}\ E]^{\pi_0}_{\pi_1,\pi_2} &\longmapsto \mathrm{RD}_{\pi_1} = \mathrm{RD}_{\pi_2} = \mathrm{RD}_{\pi_0} \\
[\textbf{select}_k]^{\pi_0}_{\pi_1,\dots,\pi_k} &\longmapsto \mathrm{RD}_{\pi_1} = \mathrm{RD}_{\pi_2} = \cdots = \mathrm{RD}_{\pi_k} = \mathrm{RD}_{\pi_0}
\end{aligned}$$

while $\forall \pi \in \mathrm{inputs}(\mathrm{N}(G)) \longmapsto \mathrm{RD}_{\pi} = \bigcup\{\, \mathrm{RD}_{\pi'} \mid \pi' \in \mathrm{pred}_G(\pi)\}$. We also assume a function $\mathrm{used}_G(\pi) = \{I_1, \dots, I_k\}$ giving, for all $\pi \in \mathrm{inputs}(\mathrm{N}(G))$, the set of local variable names *read* in any node reachable from $\pi$ (including itself).

**Definition 1.** *A Control Flow Graph $G$ is* normalized *iff, for all $[\mathbf{T}]^{\pi_0}_{\pi_1,\dots,\pi_k} \in \mathrm{N}(G)$ and $I \in \mathrm{used}_G(\pi_0)$: $|\mathrm{RD}_{\pi_0}(I)| = 1$.*

$$\frac{\mathbf{x} = [\mathbf{let}\ I = E]^{\pi_0}_{\pi_1} \in \mathrm{N}(G) \quad \mathrm{alt}(E) = \{E_1, \ldots, E_k\} \quad k > 1}{\begin{array}{c} G \rightsquigarrow G - \mathbf{x} + [\mathbf{select}_k]^{\pi_0}_{\alpha_1,\ldots,\alpha_k} + \sum_{i=1}^{k} [\mathbf{let}\ I = E_i]^{\beta_i}_{\gamma_i} + (\mathrm{pred}_G(\pi_0) \times \{\pi_0\}) \\ + \{(\alpha_i, \beta_i) \mid i = [1,k]\} + (\{\gamma_1, \ldots, \gamma_k\} \times \mathrm{succ}_G(\pi_1)) \end{array}}$$

$$\frac{\mathbf{x} = [\ell \Leftarrow E]^{\pi_0}_{\pi_1} \in \mathrm{N}(G) \quad \mathrm{alt}(\ell) = \{\ell_1, \ldots, \ell_k\} \quad \mathrm{alt}(E) = \{E_1, \ldots, E_h\} \quad k + h > 2}{\begin{array}{c} G \rightsquigarrow G - \mathbf{x} + [\mathbf{select}_{k \cdot h}]^{\pi_0}_{\alpha_{1,1},\ldots,\alpha_{k,h}} + \sum_{i=1}^{k} \sum_{j=1}^{h} [\ell_i \Leftarrow E_j]^{\beta_{i,j}}_{\gamma_{i,j}} + (\mathrm{pred}_G(\pi_0) \times \{\pi_0\}) \\ + \{(\alpha_{i,j}, \beta_{i,j}) \mid i \in [1,k],\ j \in [1,h]\} + (\{\gamma_{1,1}, \ldots, \gamma_{k,h}\} \times \mathrm{succ}_G(\pi_1)) \end{array}}$$

$$\frac{\mathbf{x} = [\mathbf{if}\ E]^{\pi_0}_{\pi_1,\pi_2} \in \mathrm{N}(G) \quad \mathrm{alt}(E) = \{E_1, \ldots, E_k\} \quad k > 1}{\begin{array}{c} G \rightsquigarrow G - \mathbf{x} + [\mathbf{select}_k]^{\pi_0}_{\alpha_1,\ldots,\alpha_k} + \sum_{i=1}^{k} [\mathbf{if}\ E_i]^{\beta_i}_{\gamma_i,\delta_i} + (\mathrm{pred}_G(\pi_0) \times \{\pi_0\}) \\ + \{(\alpha_i, \beta_i) \mid i = [1,k]\} + (\{\gamma_1, \ldots, \gamma_k\} \times \mathrm{succ}_G(\pi_1)) + (\{\delta_1, \ldots, \delta_k\} \times \mathrm{succ}_G(\pi_2)) \end{array}}$$

Fig. 8: Transformation rules for the expansion of **select** operators.

To obtain a *normalized* CFG, for all nodes $\mathbf{x} = [\mathbf{T}]^{\pi_0}_{\pi_1,\ldots,\pi_k}$ violating the above condition, namely such that $|\mathrm{RD}_{\pi_0}(I)| > 1$, two cases need to be considered.

**Case 1** This case allows duplicating one node at a time. Given $\alpha \in \mathrm{RD}_{\pi_0}(I)$, let $B_\alpha = \{\beta \in \mathrm{pred}_G(\pi_0) \mid \mathrm{RD}_\beta(I) = \{\alpha\}\}$. Then, if $\forall \beta \in \mathrm{pred}_G(\pi_0).\ |\mathrm{RD}_\beta(I)| = 1$, the following transformation can be applied:

$$G \rightsquigarrow G - (B_\alpha \times \{\pi_0\}) + \mathbf{x}' + (B_\alpha \times \{\gamma_0\}) + \textstyle\sum_{i=1}^{k} (\{\gamma_i\} \times \mathrm{succ}_G(\pi_i))$$

where $\mathbf{x}' = [\mathbf{T}]^{\gamma_0}_{\gamma_1,\ldots,\gamma_k}$ is a duplicate of $\mathbf{x}$. This transformation detaches parent $\beta$ from $\mathbf{x}$ and attaches it (as the only parent) to $\mathbf{x}'$. The output endpoints of the duplicate node are connected to the same children of $\mathbf{x}$, in the same manner. This transformation is applied in a maximal way, until its applicability condition is no longer satisfied by any node.

**Case 2** After *Case 1*, if there are still nodes for which $|\mathrm{RD}_{\pi_0}(I)| > 1$, it implies that, for some node identified by $\pi_0$, its predecessors $\mathrm{pred}_G(\pi_0)$ can be partitioned into two (non-empty) sets $B^{(1)}, B^{(2)}$ such that $\forall \beta \in B^{(1)}.\ |\mathrm{RD}_\beta(I)| = 1$, and $\forall \gamma \in B^{(2)}.\ \mathrm{RD}_\gamma(I) = \mathrm{RD}_{\pi_0}(I)$. Note that $B_\alpha \subseteq B^{(1)}$. Then the following transformation can be performed:

$$G \rightsquigarrow G - (B_\alpha \times \{\pi_0\}) + N' + E' + (B_\alpha \times \{\gamma_0\})$$

which duplicates a subgraph of $G$ which includes $\mathbf{x}$. In particular, $N'$, $E'$, $\gamma_0$ are obtained through the function $\mathrm{clone}_G(\mathbf{x}, \pi_0) = (N', E', \gamma_0)$, where: $N'$ are the duplicated nodes, $E'$ are the duplicated edges (which may also point to old nodes still present in $G$), and $\gamma_0$ is entry node of the cloned subgraph, corresponding to $\pi_0$.

In order to define the clone function, let us assume a one-to-one mapping $\alpha \mapsto \widetilde{\alpha}$ between endpoints such that $\forall \alpha, \beta.\, \alpha \neq \widetilde{\beta}$. Moreover, let us consider the subgraph $G'$ of $G$ restricted to the nodes $\mathrm{N}(G) \setminus B^{(1)}$, and let $N_0 \subseteq \mathrm{N}(G)$ be the *Strongly Connected Component (SCC)* of $G'$ containing node $\mathbf{x}$ (i.e. $\pi_0 \in \mathrm{inputs}(N_0)$). Assuming $E_\mathrm{c} = \mathrm{E}(G) \cap (N_0 \times N_0)$ and $\overline{E_\mathrm{c}} = \mathrm{E}(G) \setminus E_\mathrm{c}$, we define $\mathrm{clone}_G(\mathbf{x}, \alpha) = (N', E', \widetilde{\alpha})$, where:

```
...
int(1..2) x =
          select(1,2);
while (g < x) {
     g = (g + 1) as
          int(0..10);
}
...
```

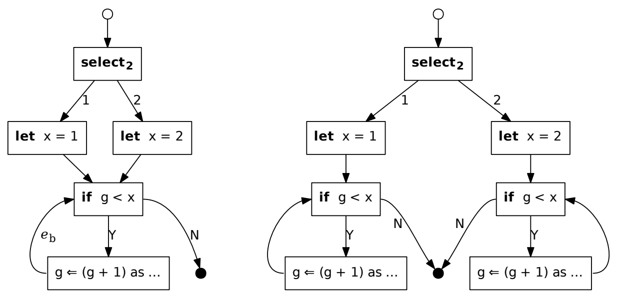


Fig. 9: Example of *while* loop duplication.

$$N' = \left\{ [\mathbf{T}]^{\widetilde{\beta_2}}_{\widetilde{\pi_1},\ldots,\widetilde{\pi_k}} \;\middle|\; [\mathbf{T}]^{\beta_2}_{\pi_1,\ldots,\pi_k} \in N_0 \wedge \exists \beta_1.\, (\beta_1, \beta_2) \in E_{\mathrm{c}} \right\};$$
$$E' = \left\{ (\widetilde{\beta_1}, \widetilde{\beta_2}) \mid (\beta_1, \beta_2) \in E_{\mathrm{c}} \right\} \cup \left\{ (\widetilde{\beta_1}, \beta_2) \mid (\beta_1, \beta_2) \in \overline{E_{\mathrm{c}}} \wedge \widetilde{\beta_1} \in \mathrm{outputs}(N') \right\}.$$

*Normalization algorithm* The algorithm alternates between the maximal application of *Case 1* and a single application of *Case 2*, until they are no longer applicable. This alternation is necessary since the transformation performed for one case may enable the application of the other case, for some nodes. Moreover, before the application of each case, the compiler performs a *constant propagation* phase, by instantiating each occurrence of any local variable with its defining expression, whenever such a definition is univocally determined. Precisely, given a node $[\mathbf{T}]^{\pi_0}_{\pi_1,\ldots,\pi_k}$, the instantiation is carried out for each variable $I$ being read in $\mathbf{T}$ such that $|\mathrm{RD}_{\pi_0}(I)| = 1$, which is replaced by an expression containing no references to local variables. At the end, a normalized CFG is obtained, that is then simplified by removing all *let* nodes, and also any *if* node for which its condition has been already reduced to a truth value.

*Example 3.* Fig. 7 shows a few transformation of CFG for the random walk model from Example 2. After the initial construction (Fig. 7a), the first transformation consists in the expansion of the **select** operator present in node [**let** $x$ = **select**$(0, 1)$], giving Fig. 7b. The original *let* node is being replaced by a *select* node followed by two new *let* nodes, one for each possible outcome 0 and 1. Then node [**if** $x$==0] is duplicated according to *Case 1* of the normalization algorithm, and the subsequent constant propagation causes both occurrences of the local variable $x$ in the *if* nodes to be instantiated, as shown in Fig. 7c. The graph obtained after the final simplification step is shown in Fig. 7d.

*Example 4.* Consider the model fragment from Fig. 9 (left), in which a variable x is non-deterministically selected and then accessed by the guard of a *while* loop. In the CFG (after the expansion of *select*) the definition of $x$ in the [**if** $g < x$] node cannot be univocally determined; in fact, both definitions of x can be traced back through the backward edge $e_{\mathrm{b}}$ coming from the end of the loop body. Hence, only Case 2 can be applied, which duplicates the SCC composed of the [**if** $g < x$] and [$g \Leftarrow (g + 1)\ldots$] nodes, yielding the CFG depicted in Fig. 9 (right).

## 4 PRISM Translation

A normalized Control Flow Graph (see Def. 1), containing only assignments to global variables $[\ell \Leftarrow E]$, *yield* nodes, *if* nodes, and *select* nodes, is taken as input for the generation of the PRISM model. In the current paper we consider only PRISM models for *Discrete Time Markov Chains* (DTMC).

**Definition 2 (PRISM syntax).** *Let Var be a countable set of variables' names. The syntax of a subset of PRISM is defined by the following grammar:*

$$\begin{array}{rcl} \mathit{PModel} & ::= & \overline{\mathit{Decl}}\ \overline{\mathit{Rule}} \\ \mathit{Decl} & ::= & \mathit{Var} : T_{\mathrm{v}}\ \textbf{init}\ v \qquad \mathit{Rule}\ ::=\ \mathit{Expr} \twoheadrightarrow (p_1 : \mathit{Upds_1} \oplus \cdots \oplus p_n : \mathit{Upds_n}) \\ \mathit{Upds} & ::= & \mathit{Upd_1}\ \&\ \ldots\ \&\ \mathit{Upd_k} \qquad \mathit{Upd}\ ::=\ \mathit{Var}' = \mathit{Expr} \\ \mathit{Expr} & ::= & v \ \big|\ \mathit{Var} \ \big|\ \mathit{Expr}\ op_{\mathrm{b}}\ \mathit{Expr} \ \big|\ op_{\mathrm{u}}\ \mathit{Expr} \ \big|\ \mathit{Expr}\ ?\ \mathit{Expr} : \mathit{Expr} \end{array}$$

*where $T_{\mathrm{v}}$, $v$, $op_{\mathrm{b}}$, $op_{\mathrm{u}}$ are as defined for Objective/MC (see Fig. 1).*

A PRISM model consists of a collection of variables $\overline{\mathit{Decl}}$, and a number of *rule schemata* $\overline{\mathit{Rule}}$ describing the transitions of the DTMC. Allowed variable types are *bounded integers* and *booleans*, analogous to the corresponding types of Objective/MC. Rules are of the form: $\mathit{guard} \twoheadrightarrow (p_1 : \mathit{alt}_1 \oplus \cdots \oplus p_n : \mathit{alt}_n)$ where *guard* is a boolean condition over the global variables determining the applicability of the rule, and $\mathit{alt}_1, \ldots, \mathit{alt}_n$ are the possible probabilistic alternatives chosen when the rule can be applied, where each $p_i \in \mathbb{R}^+$ denotes the probability of the corresponding alternative. Each alternative is composed of a collection *Upds* of *updates* of the form $\mathit{Var}' = \mathit{Expr}$, one for each global variable *Var* to be updated. Any global variable may occur at most once in the left hand side of any update in *Upds*. Moreover, rule guards must all be disjoint.

As regards the semantics, the resulting DTMC is obtained from a PRISM model by instantiating its rules for each possible state in the complete state space of the model as determined by the global variables. Since the core Objective/MC language does not currently provide operators for probabilistic choice, in the rest of the paper we assume constant probabilities among rule alternatives, namely for all rules $p_1, \ldots, p_n = 1/n$, and thus omit them from the model descriptions.

Since PRISM only allows variables to be declared as basic types (either integers or booleans), without any structure, the first step in the translation consists in flattening the Objective/MC (structured) global variables into a collection of basic PRISM variables. An array $a$ with $\mathrm{type}(a) = T[n]$ is translated into a set of PRISM variables $a[0], \ldots, a[n-1]$. Similarly, each object $o$ declared among global variables (including those in arrays) is translated into a set of PRISM variables $o.I_1, \ldots, o.I_n$, where $I_1, \ldots, I_n$ are the instance variables of $o$. Such a translation of objects into PRISM variables is possible since in Objective/MC objects cannot be created dynamically.

First of all, we determine the nodes of the CFG which give rise to transitions in the resulting PRISM model. They are the begin, assignment and yield nodes, and, together with the end node, constitute the set *States*.

We also introduce a special global variable *pc* (*program counter*) of type $\mathbf{int}(-1 \mathinner{..} N)$ where $-1 = \mathbf{err}$ denotes an *error* state used to catch violations of integral ranges allowed in expression ascriptions, and $N = |States|$. We assume a one-to-one mapping between *States* and *pc* values in $\{0, \dots, N-1\}$. For simplicity, we refer to *pc* values with their corresponding endpoints from *States*.

A PRISM rule is generated for each $\alpha \in States$. This requires determining the set of states $\text{next}(\alpha) = \{\beta_1, \dots, \beta_k\} \subseteq States$ which are *directly* reachable from $\alpha$ in the CFG, namely those states which are reachable through a path containing only *if* and *select* nodes. The generated rules have the following form, for each state $\alpha$ (except for *end* and *error*):

$$pc == \alpha \rightarrow \left( \bigoplus_{i=1}^{k} \{ \, \mathcal{U} \,\&\, (pc' = \mathcal{C}_{\text{asc}} \; ? \; \mathcal{E} : \mathbf{err}) \} \right)$$

while a *deadlock* transition $pc == \alpha \rightarrow (pc' = \alpha)$ is generated for the *end* and *error* states. As regards the first case, each alternative is composed of *(i)* a number of global assignments $\mathcal{U}$, implementing the assignment described in the current node $\alpha$, and *(ii)* an update of variable *pc*.

The different alternatives to be considered (for $i \in \{1, \dots, k\}$) emerge from the presence of *select* nodes in the subtree between $\alpha$ and the nodes in $\text{next}(\alpha)$. As regards the update ($pc' = \mathcal{C}_{\text{asc}} \; ? \; \mathcal{E} : \mathbf{err}$) of the program counter, $\mathcal{C}_{\text{asc}}$ represents the conditions obtained from the ascriptions in the subtree, and $\mathcal{E}$ is a conditional expression that evaluates to the new program counter from the set $\text{next}(\alpha)$, which is obtained from the *if* nodes present in the subtree.

*Example 5.* Consider the CFG of Fig. 7d, and let $pc = 0$ denote the *begin* node, 1 the left assignment node, and 3 the *end* node. Let $\mathcal{C}_{\text{if}} = (pos \,!\!= -50 \,\&\&\, pos \,!\!= 50)$ be the condition of the *if* node; the rules generated for nodes 0 and 1 are:

$$\begin{aligned}
pc == 0 &\rightarrow (pc' = \mathcal{C}_{\text{if}} \; ? \; 1 : 3) \oplus (pc' = \mathcal{C}_{\text{if}} \; ? \; 2 : 3) \\
pc == 1 &\rightarrow (pos' = \mathcal{C}_{\text{asc}}^{-1} \; ? \; pos - 1 : pos) \,\&\, (pc' = (\mathcal{C}_{\text{asc}}^{-1} \; ? \; (\mathcal{C}_{\text{if}}^{-1} \; ? \; 1 : 3) : \mathbf{err}) \oplus \\
&\quad\;\; (pos' = \mathcal{C}_{\text{asc}}^{-1} \; ? \; pos - 1 : pos) \,\&\, (pc' = (\mathcal{C}_{\text{asc}}^{-1} \; ? \; (\mathcal{C}_{\text{if}}^{-1} \; ? \; 2 : 3) : \mathbf{err})
\end{aligned}$$

where $\mathcal{C}_{\text{if}}^{-1}$ is used to evaluate, from the assignment node 1, the condition $\mathcal{C}_{\text{if}}$ as if the current assignment ($pos' = pos - 1$) would be already performed, while $\mathcal{C}_{\text{asc}}^{-1}$ checks the ascription in the assignment expression; formally: $\mathcal{C}_{\text{if}}^{-1} = (pos - 1 \,!\!= -50 \,\&\&\, pos - 1 \,!\!= 50)$, $\mathcal{C}_{\text{asc}}^{-1} = (-50 \leq pos - 1 \,\&\&\, pos - 1 \leq 50)$.

**Discussion** An implementation of the compiler for the core Objective/MC language into PRISM models is available [1]. We have used the compiler to automatically generate a PRISM model of the job scheduler from Fig. 2, which contains 19 transition specifications. The DTMC built by PRISM (in 0.125 seconds) consists of 256 states and 343 transitions. Verification of properties can be performed by model checking temporal logic formulas on the variables of the generated PRISM model. A hand-made PRISM model of the job scheduler example is available at [1]. It consists of 5 variables `P_0`, ..., `P_4` representing processors, a `nextjob` variable to store the value of the next processing job to be executed, and a `state` variable for the sequential execution of the steps. The model built by PRISM in this case (in 0.035 seconds) consists of 264 states and 312 transitions.

In terms of DTMC dimension, the PRISM model obtained from the translation of the Objective/MC model turns out to be similar in size to the hand-made PRISM models. This suggests that the pruning of local variables performed at compile time has actually avoided too many useless states to be introduced by the translation. Note also that in the hand-made PRISM model, changing the range of possible values for the next processing job to be executed requires manually modifying a number of transition specifications in the model source code. Instead, in the Objective/MC model in Fig. 2, the same modification could be done by simply changing the value of the `select` expression at line 12.

## 5 Conclusions

We have proposed an early version of an object-oriented language, called Objective/MC, for the specification of models to be analysed by model checking. In particular, we have focused on the imperative constructs of the language and on the handling of global and local variables. As future developments, we plan to formally define the semantics of the language and prove the correctness of the transformations performed on the models. Moreover, we will extend the language with richer object oriented features (methods and inheritance), with richer data structures and operations on them (array filters and a notion of graph) and with probabilistic/stochastic operations. Finally, we plan to add some features to deal with concurrency aspects.

## References

1. ObjMC: The Objective/MC compiler, `http://pages.di.unipi.it/pardini/ObjMC`
2. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Computer Aided Verification. pp. 359–364. Springer (2002)
3. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Alg. for the Construction and Analysis of Systems, pp. 168–176. Springer (2004)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science 285(2), 187–243 (2002)
5. Havelund, K., Pressburger, T.: Model checking Java programs using Java Pathfinder. Int. Journal on Software Tools for Technology Transfer 2(4), 366–381 (2000)
6. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on software engineering 23(5), 279 (1997)
7. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Computer aided verification. pp. 585–591. Springer (2011)
8. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1(1), 134–152 (1997)
9. Sirjani, M., Movaghar, A., Shali, A., De Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informaticae 63(4), 385–410 (2004)