Generating compact MBTDD-representations from **Probmela** specifications

Frank Ciesinski[1], Christel Baier[1], Marcus Größer[1],
David Parker[2]

[1]Technical University Dresden, Institute for Theoretical Computer Science, Germany
[2]Oxford University Computing Laboratory, Oxford, UK

Abstract. The purpose of the paper is to provide an automatic transformation of parallel programs of an imperative probabilistic guarded command language (called Probmela) into probabilistic reactive module specifications. The latter serve as basis for the input language of the symbolic MTBDD-based probabilistic model checker PRISM, while Probmela is the modeling language of the model checker LiQuor which relies on an enumerative approach and supports partial order reduction and other reduction techniques. By providing the link between the model checkers PRISM and LiQuor, our translation supports comparative studies of different verification paradigms and can serve to use the (more comfortable) guarded command language for a MTBDD-based quantitative analysis. The challenges were (1) to ensure that the translation preserves the Markov decision process semantics, (2) the efficiency of the translation and (3) the compactness of the symbolic BDD-representation of the generated PRISM-language specifications.

1 Introduction

Model checking plays a crucial role in analyzing quantitative behaviour of a wide range of system types such as randomised distributed algorithms and randomised communication protocols. One of the key ingredients of a model checking tool for a randomized system is an appropriate modeling language which should be expressive and easy to learn and must be equipped with a formal semantics that assigns MDPs to the programs of the modeling language. For efficiency reasons, it is also important that the MDP-semantics has a formalization by means of rules that support the automated generation of a compact internal representation of the MDP from a given program. The modeling languages of most model checkers for MDPs use probabilistic variants of modeling languages of successful non-probabilistic model checkers. The MDP-fragment of the model checker PRISM [8] uses reactive module-like specifications [1] extended by the feature that statements can have a probabilistic effect. Probabilistic reactive modules rely on a declaration of pre- and postconditions of variables. Thus, their nature is rather close to symbolic representations which makes

them well suited for the generation of a multiterminal binary decision diagram (MTBDD) [6, 19] for the system. On the other hand, probabilistic reactive modules do not support complex data structures (e.g., arrays and channels) and require the encoding of conditional or repetitive commands by means of pre- and postconditions. Modeling languages, like Probmela [2] which is a probabilistic dialect of the prominent (nonprobabilistic) modeling language Promela [9], that combine features of imperative programming language (such as complex datatypes, conditional commands, loops) with message passing over channels and communication over shared variables are much more comfortable. Many protocols and systems can be formally described within such a high-level modeling language in a rather elegant and intuitive way. The MDP-semantics of a given Probmela-program can be obtained as a DFS-based enumeration of the reachable states (similar to the on-the-fly generation of the transition system for a given Promela-program as it is realized in SPIN [9]). However, the generation of a compact symbolic MTBDD-representation is nontrivial. Although several reduction techniques that rely on an analysis of the underlying graph of the MDP or the control graphs of Probmela-programs can be applied to make the quantitative analysis competitive with the symbolic approach concerning the time required for the quantitative analysis [5], the enumerative approach often fails to handle very large systems with many parallel processes which can still be verified with the symbolic approach by PRISM.

The purpose of this paper is to combine the advantages of both approaches by providing an automatic translation from Probmela-programs into PRISM language and to derive a compact MTBDD representation from the generated PRISM code. The implementation of this translation (called Prismela) yields the platform to use the (more comfortable) modeling language Probmela for the MTBDD-based quantitative analysis of PRISM and supports comparative studies of different verification paradigms: the symbolic approach realized in PRISM [8] and the enumerative approach of LiQuor [4]. The main challenges are

(1) to ensure that the translation preserves the Markov decision process semantics, (without introducing extra steps and intermediate pseudo-states that serve to simulate a single step of the original guarded command specification),
(2) the efficiency of the translation and
(3) the compactness of the symbolic BDD-representation of the generated PRISM-module specifications.

Our work is conceptually related to [3], where a translation schema is presented that allows for the transformation of a core fragment of Promela to the input format of the (nonprobabilistic) symbolic model checker

SMV[14]. Beside the probabilistic features (probabilistic choice, lossy channels, random assignments), we treat some more language concepts than [3] such as message passing via handshaking through synchronous channels. Furthermore, we describe the translation of atomic regions in more detail and describe our implemented automated heuristics to calculate a good variable ordering for a given model. Another symbolic approach for Promela specifications has been presented in [20] using a nonstandard decision diagram, called DDD.

After a brief summary of the main concepts of Probmela and the PRISM input language (Section 2), we present the translation (Section 3), discuss heuristics that address item (3) and attempt finding good variable orderings for the MTBDD-representation and determining the variable ranges (Section 4). In Section 5, we explain the main features of our implementation on the top of the model checkers LiQuor and PRISM and report on experimental results.

2 Preliminaries

We give here brief intuitive explanations on the syntax and semantics of the (core fragment of the) modeling language Probmela and PRISM's language, and suppose that the reader is familiar with the main concepts of Promela [9] and reactive modules [1].

The modeling language Probmela *[2]* is a probabilistic dialect of SPIN's input language Promela [9]. In the core language, programs are composed by a finite number of processes that might communicate over shared (global) variables or channels. Programs consist of a declaration (types, initial values) of the global variables and channels, and the code for the processes. The processes can access the global variables and channels, but they also can have local variables and channels. We skip these details here and suppose for simplicity that the names of all (local or global) variables and channels are pairwise distinct. The channels can be synchronous or fifo-channels of finite capacity. The fifo channels can declared to be either perfect or lossy with some failure probability $\lambda \in]0,1[$. The meaning of λ is that the send-operation might fail with probability λ. The operational behavior of the processes is specified in a guarded command language as in Promela with with (deterministic) assignments $x = \mathsf{expr}$, communication actions $c?x$ (receiving a value for variable x along channel c) and $c!\mathsf{expr}$ (sending the current value of an expression along channel c), the statement skip, conditional and repetitive statements over guarded commands (`if`...`fi` and `do`...`od`), and atomic regions. The probabilisic features of Probmela are lossy fifo-channels (see above), a probabilistic choice operator $\texttt{pif}[\pi_1] \Rightarrow \mathsf{cmd}_1 \ldots [\pi_k] \Rightarrow \mathsf{cmd}_k\texttt{fip}$ (where $\pi_1, \ldots, \pi_k$ are

probabilities, i.e., real numbers between 0 and 1 such that $\pi_1+\ldots+\pi_k \leq 1$ and $\mathsf{cmd}_1, \ldots, \mathsf{cmd}_k$ are Probmela commands) and random assignments $x = \mathsf{random}(V)$. The intuitive meaning of the `pif ... fip` command is that with probability π_i, command cmd_i is executed next. The value $1-(\pi_1+\ldots+\pi_k)$ is the deadlock probability where no further computation of the process is possible. In a random assignment $x = \mathsf{random}(V)$, x is a variable and V a finite set of possible values for x. The meaning is that x is assigned to some value in V according to the uniform distribution over V. In addition, Probmela permits jumps by means of goto-statements.

Probmela also supports the creation, stopping, restarting and destruction of processes. Since the PRISM language assumes a fixed number of variables and modules, such dynamic features are not included in the translation and are therefore irrelevant for the purposes of this paper.

PRISM*'s input language [17].* For the purposes of the paper, only the fragment of PRISM that has an MDP-semantics is relevant. In this fragment, a PRISM program consists of several *modules* $\mathcal{P} = Q_1 \| ... \| Q_n$ that run in parallel. Each module consists of a variable definition and a finite set of statements. The statements are equipped with a precondition (guard) on the current variable evaluation. The effect of the statements on the variables can be probabilistic. A PRISM statement $s \in \mathsf{Stmt}$ has the form

$$[\sigma]\,\mathsf{guard} \;\rightarrow\; \pi_1 : \mathsf{upd}_1 + ... + \pi_k : \mathsf{upd}_k$$

where guard is a Boolean condition on the variables and $\pi_1, \ldots, \pi_k$ are probabilities, i.e., real numbers between 0 and 1 that sum up to 1. (If $k = 1$ then $\pi_1 = 1$ and we simply write $[\sigma]\mathsf{guard} \rightarrow \mathsf{upd}_1$ rather than $[\sigma]\mathsf{guard} \rightarrow 1 : \mathsf{upd}_1$.) The terms upd_i are "updates" that specify how the new values of the variables are obtained from the current values. Formally, the updates are conjunctions of formulas of the type $x' = \mathsf{expr}$ where x is a program variable and its primed version x' refers to the value of x in the next state and expr is an expression built by constants and (unprimed) variables. If an update does not contain a conjunct $x' = \ldots$ then the meaning is that the value of variable x remains unchanged. (In this way, the updates upd_i specify unique next values.) The symbol σ is either ε or a synchronization label. Statements of different modules with the special symbol ε (simply written [ ] rather than $[\varepsilon]$) are executed in an interleaved way, i.e., without any synchronization. The meaning of statements with a synchronization label σ is that all modules have to synchronize over a statements labeled by σ. No other channel-based communication concept is supported by the PRISM language, i.e., there is no asynchronous message passing over fifo-channels and no (explicit) operator modeling handshaking between two modules. Furthermore, PRISM does not support data types like arrays.

Markov decision processes (MDP). Both Probmela programs and PRISM programs have an operational semantics in terms of a Markov decision process (MDP) [18]. In this context, the MDP for a program consists of a finite state space S and a transition relation $\rightarrow \subseteq S \times Act \times Distr(S)$ where Act is a set of actions and $Distr(S)$ denotes the set of (sub) distributions over S (i.e., functions $\mu : S \rightarrow [0,1]$ such that $\sum_{s \in S} \mu(s) \leq 1$). Furthermore, there is a distinguished state that is declared to be initial.

The states in the MDP for a Probmela program consist of local control states for all processes, valuations for the local and global variables and a component that specifies the current contents of the fifo-channels. The transition relation $\rightarrow$ is formally presented by means of SOS-rules [2]. In our implementation, we slightly departed from [2] and used a MDP-semantics that relies on a representation of each process by a control graph, which can then be unfolded into an MDP and put in parallel with the MDPs for the other processes. (Parallel composition is understood as ordinary interleaving and synchronization in the handshaking principle for message passing over synchronous channels.)

In the sequel, let Var be the set of all global variables of the given program and $LocVar_i$ the set of local variables of process Q_i. For simplicity, we suppose that $Var \cap LocVar_i = \emptyset$. We write Var_i for $Var \cup LocVar_i$, the set of variables that can appear in the statements of process Q_i. If V is a set of variables then $Eval(V)$ denotes the set of all (type-consistent) valuations for the variables in V. In the control graph for process Q_i, the nodes are called locations of Q_i. They play the role of a program counter and are obtained by assigning identifiers to each command in the Probmela-code for Q_i. The edges have the form $\ell \overset{g:\alpha}{\rightsquigarrow} \nu$ where ℓ is a location, g is a guard (Boolean condition on the variables in Var_i) and α an action which can be viewed as a function $\alpha : Eval(Var_i) \rightarrow Distr(Eval(Var_i))$ and ν a distribution over the locations of Q_i. If ν assigns probability 1 to some location ℓ' (and probability 0 to all other locations) then we simply write $\ell \overset{g:\alpha}{\rightsquigarrow} \ell'$. Furthermore, the trivial guard $g = \text{true}$ is omitted and we simply write $\ell \overset{\alpha}{\rightsquigarrow} \nu$ rather than $\ell \overset{true:\alpha}{\rightsquigarrow} \nu$. For instance, the location ℓ assigned to the command $\texttt{pif}[\pi_1] \Rightarrow \mathsf{cmd}_1 \ldots [\pi_k] \Rightarrow \mathsf{cmd}_k\texttt{fip}$ has just one outgoing edge $\ell \overset{id}{\rightsquigarrow} \nu$ where id is the identity[1]. Let ℓ_j be the location representing the command cmd_j then distribution ν is given by $\nu(\ell_j) = \pi_j$ and $\nu(\ell') = 0$ for all other locations ℓ'. If location ℓ stands for a nondeterministic choice $\texttt{if}[g_1] \Rightarrow \mathsf{cmd}_1 \ldots [g_k] \Rightarrow \mathsf{cmd}_k\texttt{fi}$, then there are k outgoing edges $\ell \overset{g_j:id}{\rightsquigarrow} \ell_j$. Similarly, for a loop $\texttt{do}[g_1] \Rightarrow \mathsf{cmd}_1 \ldots [g_k] \Rightarrow \mathsf{cmd}_k\texttt{od}$, there are k outgoing edges $\ell \overset{g_j:id}{\rightsquigarrow} \ell_j$ for $1 \leq j \leq k$ where ℓ_j is

[1] i.e., $id(\eta)(\eta) = 1$ and $id(\eta)(\eta') = 0$ if $\eta \neq \eta'$

the location representing cmd_j.[2] If ℓ represents an assignment $x = \mathsf{expr}$, then ℓ has a single outgoing edge $\ell \overset{true:\alpha}{\rightsquigarrow} \ell'$ with the trivial guard true and the action α that modifies x according to expr and keeps all other variables unchanged. Again, location ℓ' stands for the command after the command represented by ℓ in the Probmela code for Q_i. The effect of an atomic regions is modeled in the control graph by a single edge that represents the cumulative effect of all activities inside the atomic region.

For a given PRISM program, the MDP is obtained as follows. The states are the evaluations of the program variables. Given a state s, then for each statement $\mathsf{stmt} = [\,]\mathsf{guard} \to \pi_1 : \mathsf{upd}_1 + \ldots + \pi_k : \mathsf{upd}_k$ (in some module) where $s \models \mathsf{guard}$ there is a transition $s \overset{\mathsf{stmt}}{\to} \nu$ with $\nu(s') = \sum_j \pi_j$ where j ranges over all indices in $\{1, \ldots, k\}$ such that upd_j evaluates to true when the unprimed variables are interpreted accoding to s, while the values of the primed variables are given by s'. Furthermore, s and s' must agree on all variables x where x' does not appear in upd_j.

3 From Probmela to PRISM

We now suppose that we are given a Probmela program $\mathcal{P}$ (of the core language without dynamic features). The goal is to generate automatically a PRISM program $\tilde{\mathcal{P}}$ that has the same MDP-semantics and a compact MTBDD-representation. The general workflow of the translation (see Fig. 1) starts with a Probmela program $\mathcal{P}$ consisting of n processes $Q_1, \ldots, Q_n$ and derives a PRISM program $\tilde{\mathcal{P}}$ with n modules $\tilde{Q}_1, \ldots, \tilde{Q}_n$. The global variables of $\mathcal{P}$ are also global in $\tilde{\mathcal{P}}$. Furthermore, $\tilde{\mathcal{P}}$ contains additional global variables that serve to mimic the arrays and channels in $\mathcal{P}$ and other features of Probmela that have no direct translation. The last two steps attempt to minimize the MTBDD-representation and rely on heuristics to determine a good variable ordering and algorithms that fix appropriate bit-sizes (ranges) of variables (Section 4).

In the first step, each Probmela process is translated into an equivalent PRISM module. It relies on the control graph semantics of Probmela and translates each control edge into one or more PRISM statements. Given a Probmela process Q_i, the corresponding PRISM module $\tilde{Q}_i$ has the same local variables $LocVar_i$ and an additional integer variable pc_i that serves as a program counter for Q_i. Intuitively, the possible values of pc_i encode the locations of the control graph of Q_i. The rough idea is to replace each edge $\ell \overset{g:\alpha}{\rightsquigarrow} \nu$ in the control graph of Q_i with the PRISM statement

$$[\,](\tilde{g} \wedge pc_i = \ell) \;\to\; \nu(\ell_1) : (\tilde{\alpha}) \wedge (pc_i = \ell_1) \;+ \ldots + \nu(\ell_k) : (\tilde{\alpha}) \wedge (pc_i = \ell_k)$$

[2] As in Promela loops are terminated by a special command `break`. Its control semantics is given by a control edge from the location of the break-command to the next location after the loop.

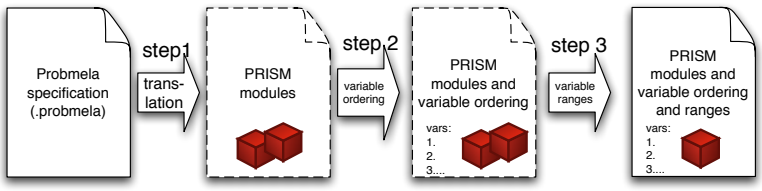

Fig. 1. Translation scheme from Probmela to PRISM

where $\tilde{g}$ and $\tilde{\alpha}$ are the translations of g and α to PRISM code with primed and unprimed variables and $\ell_1, \ldots, \ell_k$ are the locations that have positive probability under distribution ν.

This basic translation schema is directly applicable for deterministic or randomized assignments. For instance, if α stands for the assignment $x = y+z$ then $\tilde{\alpha}$ is the update $(x' = y+z)$. Since PRISM and Probmela support the same operations on basic types, the translation of Boolean conditions and actions representing simple assignments is straightforward, even when they involve more complex operations like multiplication, division, etc. Thus, a deterministic assignment given by an edge $\ell \overset{\alpha}{\rightsquigarrow} \ell'$ in the control graph where α is given by the command $x = \mathsf{expr}$ is translated into the PRISM statement $[\,]pc_i = \ell \;\rightarrow\; (x' = \mathsf{expr}) \wedge (pc_i' = \ell')$. For a randomized assignment, given by an edge $\ell \overset{\alpha}{\rightsquigarrow} \ell'$ where α is given by the command $x = \mathsf{random}(0, 1)$, the translation schema yields the PRISM statement

$$[\,]pc_i = \ell \;\rightarrow\; \tfrac{1}{2} : (x' = 0) \wedge (pc_i' = \ell') \;+\; \tfrac{1}{2} : (x' = 1) \wedge (pc_i' = \ell').$$

Arrays are very useful to model, e.g. memory slots or network packages. Probmela uses a C-like syntax to define arrays (e.g., `int`$[3]a$ defines an integer array with 3 cells). Access to the array cells is possible either using a variable, a constant or an arithmetical expression as an index, e.g. $a[i + j]$. Arrays are also supported in classical reactive modules [1], but they are difficult to implement if a reactive module specification has to be transfered into symbolic representation and are therefore not supported in PRISM. For an action α which contains an array access of the form $a[\mathsf{expr}]$ the corresponding condition $\tilde{\alpha}$ is obtained by introducing fresh PRISM variables a_j for all array cells $a[j]$ and replacing $a[j]$ by a_j or a_j' (depending on whether $a[j]$ appears on the left or right hand side of an assignment). A more complex case occurs if the array index to which is referenced is an array access itself. For instance, if action α in the control edge $\ell \overset{\alpha}{\rightsquigarrow} \ell'$ is the assignment $a[j[k]] = 7$ then the corresponding PRISM code consists of several statements that represent the possible combinations of values for k and $j[k]$:

$$
\begin{array}{l}
[\,](pc_i = \ell) \wedge (k = 0) \wedge (j_0 = 0) \rightarrow (a'_0 = 7) \wedge (pc'_i = \ell') \\
[\,](pc_i = \ell) \wedge (k = 0) \wedge (j_0 = 1) \rightarrow (a'_1 = 7) \wedge (pc'_i = \ell') \\
[\,](pc_i = \ell) \wedge (k = 0) \wedge (j_0 = 2) \rightarrow (a'_2 = 7) \wedge (pc'_i = \ell') \\
\vdots \\
[\,](pc_i = \ell) \wedge (k = 1) \wedge (j_1 = 0) \rightarrow (a'_0 = 7) \wedge (pc'_i = \ell') \\
[\,](pc_i = \ell) \wedge (k = 1) \wedge (j_1 = 1) \rightarrow (a'_1 = 7) \wedge (pc'_i = \ell') \\
[\,](pc_i = \ell) \wedge (k = 1) \wedge (j_1 = 2) \rightarrow (a'_2 = 7) \wedge (pc'_i = \ell') \\
\vdots
\end{array}
$$

The case where multiple arrays are connected via arithmetical operations, e.g. $a[j[k] + l[m]] = 42$, can be treated in a similar way.

Remark 1. The treatment of probabilistic choices (`pif` ... `fip`), nondeterministic choices (`if` ... `fi`), loops (`do` ... `od`) and jumps is inherent in our translation schema which operates on the control graph semantics of Probmela (and where the meaning of probabilistic, nondeterminstic choices and loops is already encoded). However, it is worth noting that our translation yields a rather natural and intuitive encoding in PRISM for these language concepts. The translation of a probabilistic choice $\texttt{pif}[\pi_1] \Rightarrow \mathsf{cmd}_1 \ldots [\pi_k] \Rightarrow \mathsf{cmd}_k\texttt{fip}$, specified by an edge $\ell \overset{id}{\rightsquigarrow} \nu$ (where $\nu(\ell_j) = \pi_j$ and ℓ_j is the location for cmd_j) yields the PRISM statement

$$[\,](pc_i = \ell) \;\rightarrow\; \pi_1 : (pc'_i = \ell_1) \;+ \ldots + \pi_k : (pc'_i = \ell_k).$$

For a nondeterministic choice $\texttt{if} :: g_1 \Rightarrow \mathsf{cmd}_1 \ldots :: g_k \Rightarrow \mathsf{cmd}_k\texttt{fi}$ or loop $\texttt{do} :: g_1 \Rightarrow \mathsf{cmd}_1 \ldots :: g_k \Rightarrow \mathsf{cmd}_k\texttt{od}$ specified by k control edges $\ell \overset{g_j:id}{\rightsquigarrow} \ell_j$ we get k PRISM statements $[\,](\tilde{g}_j) \wedge (pc_i = \ell) \rightarrow (pc'_j = \ell_j)$ for $1 \leq j \leq k$. Similarly, the basic translation schema can directly be applied to treat Probmela's goto-command, formalized by control edges of the form $\ell \overset{g}{\rightsquigarrow} \ell'$ for a conditional jump. The corresponding PRISM statement has the form $[\,](\tilde{g}) \wedge (pc_i = \ell) \rightarrow (pc'_i = \ell')$. □

Perfect asynchronous channels can be regarded as arrays with restricted access according to the fifo principles. In the internal representation of our model checker LiQuor, asynchronous channels are realized as arrays with an additional variable c_{fill} that keeps track of the number data items currently stored inside the channel. That is, $c_{\text{fill}} = k$ iff channel c contains k messages. A send operation $c!v$ is enabled iff c_{fill} is strictly smaller than the capacity of c (which is defined in the channel declaration), while a receive operation $c?x$ requires $c_{\text{fill}} > 0$. When executing a send or receive operation, variable c_{fill} is incremented or decremented, respectively. The translation of send and receive operations into PRISM statements can therefore be realized in a similar way as array access. E.g., the control

edge $\ell \overset{g:c!v}{\rightsquigarrow} \ell'$ where c is a perfect channel of capacity m and v a constant value is translated into the PRISM statements:

$$[\,](pc_i = \ell) \wedge (\tilde{g} \wedge c_{\text{fill}} = j - 1) \rightarrow \\ (pc_i' = \ell') \wedge (c_{\text{fill}}' = j) \wedge (c_j' = c_{j-1}) \wedge \ldots (c_2' = c_1) \wedge (c_1' = v),$$

while control edges $\ell \overset{g:c?x}{\rightsquigarrow} \ell'$ representing a receive operation are realized in PRISM by the statements

$$[\,](pc_i = \ell) \wedge (\tilde{g}) \wedge (c_{\text{fill}} = j) \rightarrow (pc_i' = \ell') \wedge (c_{\text{fill}}' = j - 1) \wedge (x' = c_j)$$

where $1 \leq j \leq m$. The shift operation that is inherent in the PRISM code for the send operation serves to avoid that the same channel configuration is presented by several states.

Lossy asynchronous channels In Probmela, asynchronous channels can be declared to be lossy, i.e. the enqueueing process loses the message with some predefined probability. For such lossy channels, we modify the translation for perfect asynchronous channels by dealing with a probabilistic choice for the PRISM statement modeling the send operation. Suppose that the send operation of process Q_i is modeled by the control edge $\ell \overset{c!v}{\rightsquigarrow} \ell'$ and that the failure probability of c is 0.3. The corresponding PRISM statements are:

$$\begin{array}{ll} [\,](pc_i = \ell) \wedge (c_{\text{fill}} = 2) & \rightarrow 0.7 : (c_{\text{fill}} = 3) \wedge (c_2' = c_1) \wedge (c_1' = c_0) \wedge (c_0' = v) \wedge (pc' = \ell') + \\ & \quad 0.3 : (pc' = \ell) \\ [\,](pc_i = \ell) \wedge (c_{\text{fill}} = 1) & \rightarrow 0.7 : (c_{\text{fill}} = 2) \wedge (c_1' = c_0) \wedge (c_0' = v) \wedge (pc' = \ell') + \\ & \quad 0.3 : (pc' = \ell) \\ [\,](pc_i = \ell) \wedge (c_{\text{fill}} = 0) & \rightarrow 0.7 : (c_{\text{fill}} = 1) \wedge (c_0' = v) \wedge (pc' = \ell') + 0.3 : (pc' = \ell) \end{array}$$

Synchronous channels are syntactically defined in Probmela (as well in Promela) as channels with capacity 0. They require pairwise message passing by handshaking between processes. If c is a synchronous channel then the send operation $c!v$ can only be performed if there is another process ready to immediately execute a receive operation $c?x$. The PRISM language also supports synchronization, but without message passing and not in a pairwise manner. Instead, synchronization in PRISM language modules is over the synchronization labels and requires the participation of all modules. To translate Probmela's communication actions for a synchronous channel c into PRISM code, we generate appropriate synchronization labels all potential handshakings along channel c. That is, for each pair of control edges $e_1 = \ell_i \overset{c?x}{\rightsquigarrow} \ell_i'$ in the control graph of process Q_i and $e_2 = \ell_j \overset{c!\mathsf{expr}}{\rightsquigarrow} \ell_j'$ in the control graph of another process Q_j with matching handshaking actions we introduce a fresh synchronization label $\sigma(e_1, e_2)$ and use the following PRISM statements:

$$
\begin{array}{ll}
[\sigma(e_1,e_2)]\ (pc_i = \ell_i) \ \rightarrow (pc_i' = \ell_i') \wedge (x' = \mathsf{expr}) & \text{(in module } \tilde{Q}_i\text{)} \\
[\sigma(e_1,e_2)]\ (pc_j = \ell_j) \ \rightarrow (pc_j' = \ell_j') & \text{(in module } \tilde{Q}_j\text{)} \\
[\sigma(e_1,e_2)]\ (pc_k = \ell_k) \rightarrow (pc_k' = \ell_k) & \text{(in module } \tilde{Q}_k,\ k \notin \{i,j\}\text{)}
\end{array}
$$

Note that the use of synchronization labels $c_{i,j}$ that just indicate the channel and synchronization partners would not be sufficient since a process might request a synchronous communication actions at several locations.

Atomic regions collapse several commands to one single step. They can be used to effectively shrink the state space if it is known that certain calculations need not (or must not) to be carried out interleaved. To the user this language element appears as a builtin mutual exclusion protocol that can be used to execute certain calculations exclusively without actually implementing a mutual exclusion mechanism as part of the specification. In the simple case, the atomic region consists of a sequential composition of independent assigments, e.g., $a = 1; b = 2; c = 3$, which corresponds to the PRISM statement $[\,]\text{true} \rightarrow (a' = 1) \wedge (b' = 2) \wedge (c' = 3)$. However, more complicated types of atomic regions that are allowed in Probmela that require a more involved translation into PRISM. First, atomic regions can write a single variable more than once. Consider for instance the atomic region $\texttt{atomic}\{i{+}{+}; i{+}{+}\}$ which would (according to the simple translation scheme) lead to an update $...- > (i' = i{+}1) \wedge (i' = i{+}1)$. Such statements, however, are not allowed in PRISM. Instead, such commands must either be subsumed to one expression (i.e. $i' = i + 2$) or encoded in two separate transitions. Second, atomic regions may contain (nested) probabilistic or nondeterministic choices that can hardly be accumulated into a single step. To provide the PRISM code for complex atomic regions, an additional global variable *proc* is added to the PRISM program $\tilde{\mathcal{P}}$. We set *proc* to an initial value of -1 and extend the guards of PRISM statements in each module $\tilde{Q}_i$ by the condition $proc = -1 \vee proc = i$. This ensures that for $proc = -1$ all modules can potentionally perform steps, while for $proc = i$ the transitions of all module $\tilde{Q}_j$ with $j \neq i$ are disabled. Furthermore, we extend the PRISM code for module $\tilde{Q}_i$ to ensure that when an atomic region of process Q_i is entered then the current value of *proc* is set to i and that *proc* is reset to -1 when Q_i leaves the atomic region.

Soundness of the translation. The reachable fragments of the MDPs for a given Probmela program $\mathcal{P}$ without atomic regions and the generated PRISM program $\tilde{\mathcal{P}}$ are *isomorphic*. The isomorphism is obtained by identifying the state $s = \langle \ell_1, \ldots, \ell_n, \eta \rangle$ in the MDP for $\mathcal{P}$ with the state $s' = \langle pc_1 = \ell_1, \ldots, pc_n = \ell_n, \tilde{\eta} \rangle$ in the MDP for $\tilde{\mathcal{P}}$. Here, ℓ_i is a location in the control graph for process Q_i and η a variable and channel valuation. $\tilde{\eta}$ stands for the unique valuation of the variables in $\tilde{\mathcal{P}}$ that is consistent

with η (i.e., agrees on all variables of $\mathcal{P}$ and maps, e.g., the index-variables a_j for an array a in $\mathcal{P}$ to the value of the j-th array cell $a[j]$ under η). To show that each outgoing transition has a matching transition from s', and vice versa, we can make use of the fact that the outgoing transition from both s and s' arise by the control edges from the locations ℓ_i and that the PRISM statements are defined exactly in the way such that the enabledness and the effect of the control edges is preserved. This strong soundness result still holds if $\mathcal{P}$ contains simple atomic regions. In case that $\mathcal{P}$ contains complex atomic regions then we can establish a *divergence-sensitive branching bisimulation* between the (reachable fragments of the) MDPs for $\mathcal{P}$ and $\tilde{\mathcal{P}}$ which identifies all (intermediate) states where the location of some process is inside an atomic regions. Thus, $\mathcal{P}$ and $\tilde{\mathcal{P}}$ are still equivalent for all stutter-insensitive properties, e.g., specified by nextfree LTL or PCTL formulae.

4 Optimizations of the MTBDD representation

The translation presented in the previous section combined with the PRISM tool yields an automatic way to generate a symbolic representation of the MDP for a Probmela program as a multiterminal binary decision diagram (MTBDD) [6, 19]. In this section, we discuss techniques to obtain a compact MTBDD representation. First, we present a heuristic to find a good variable ordering for the MTBDD of a given PRISM program. Second, we address the problem of finding appropriate and small ranges for the variables in a PRISM program.

Determining good variable orderings automatically Throughout this section, we assume some familiarity with (MT)BDDs. (Details can be found, e.g., in [15, 21].) It is well-known that the size of an (MT)BDD for a discrete function can crucially depend on the underlying variable ordering and that the problem of finding the optimal variable ordering is NP-complete. There are several heuristic approaches to find fairly good variable orderings. Some of them improve the variable ordering of a given (MT)BDD, while others attempt to derive a good initial variable ordering from the syntactic description of the function to be represented [7, 16]. We follow here the second approach and aim to determine a reasonable variable ordering from the PRISM code.

Given a PRISM program we abstract away from the precise meaning of Boolean or arithmetic operations and analyze the dependencies of variables. For this, we treat the PRISM statements as statements that access variables by means of uninterpreted guards and operations. This leads to an *abstract syntax tree* (AST) presenting the syntactic structure of the given PRISM program $\tilde{\mathcal{P}}$. For this, we regard the PRISM statements as

terms over the signature that contains constant symbols and the primed and unprimed versions of the program variables as atoms and uses symbols like $+, *, =, <, \rightarrow$ as function symbols. (The probabilities attached to updates are irrelevant and can simply be ignored). The node set in the AST for $\tilde{\mathcal{P}}$ consists of all statements in $\tilde{\mathcal{P}}$ and their subterms the primed and unprimed versions of the variables of $\tilde{\mathcal{P}}$ and nodes for all function symbols that appear in the statements of $\tilde{\mathcal{P}}$ (like comparison operators, arithmetic operators, the arrows between the guard and sum of updates in statements). Furthermore, the AST contains a special root node δ that serves to link all statements. The edge relation in the AST is given by the "subterm relation". That is, the leaves stand for the primed or unprimed variables or constants.[3] The children of each inner node v represent the maximal proper subterms of the term represented by node v. The children of the root note are the nodes representing the statements.

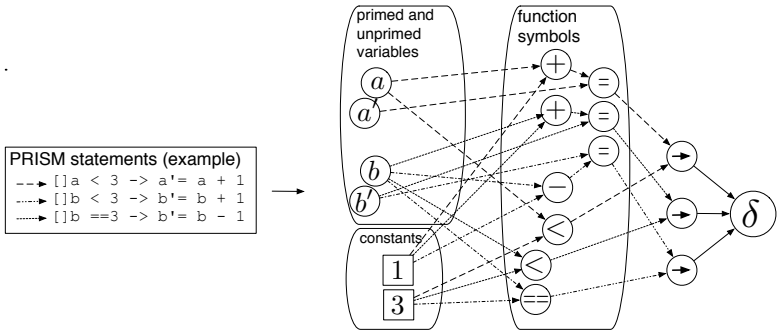

Fig. 2. Example AST.

We now apply simple graph algorithms to the AST of $\tilde{\mathcal{P}}$ to derive a reasonable variable ordering for the MTBDD for $\tilde{\mathcal{P}}$. For this, we adapt heuristics that have been suggested for gate-level circuit representations of switching functions. We considered the fanin-heuristic [13] and the weight-heuristic [10] and adapted them for our purposes. The rough idea behind these heuristics is to determine a variable ordering such that (1) variables that affect the program at most should appear at the top levels, and (2) variables that are near to one another in the dataflow should be grouped together.

The *fanin-heuristic* is based on the assumption that input variables that are connected to the output variables via longer paths are more meaningful to the function and should be ordered first. For this a breadth-first-search is performed (starting from the leaves in the AST, i.e., the

[3] At the bottom level, leaves representing the same variable or constant are collapsed. So, in fact, the AST is a directed acyclic graph, and possibly not a proper tree.

variables and constant symbols) which labels all nodes of the graph with the maximum distance to an input node, i.e., we compute the values $d(v)$ for all nodes in the AST where $d(v) = 0$ for the leaves and

$$d(w) = 1 + \max\{d(v) : v \text{ is a child of } w\}$$

for all inner nodes w. The second step of the heuristic performs a depth-first-search starting at the root node with the additional property that the depth-first-search order in each node w that is visited is according to a descending ordering of the values $d(v)$. The visiting order of the variables then yields a promising variable ordering for the MTBDD for $\tilde{\mathcal{P}}$.

The *weight-heuristic* relies on an iterative approach that assigns weights to all nodes of the AST and in each iteration the variable with the highest weight is the next in the variable ordering. This variable as well as any node that cannot reach any other variable is then removed from the AST and the next iteration yields the next variable in the ordering. (We suppose here that initially the leaves representing constants are removed from the AST.) In each iteration the weights are obtained as follows. We start with the root node and assign weight 1 to it and then propagate the weight to the leaves by means of the formula:

$$\text{weight}(v) = \frac{\text{weight}(\text{father}(v))}{|\text{number of children of father(v)}|}$$

Determining variable ranges Besides the variable ordering, the bitsizes (and hence the value ranges) of the variables in a specification have great influence on the size of the MTBDD. This is unfortunately even the case if it turns out during model construction that in the reachable part of the model a particular variable does not fully exploit its defined range. Thus, it is highly desirable that the variable ranges in the PRISM model are as "tight" as possible. Often the user does this by applying her/his knowledge about the model and choosing just a reasonable range for each variable. Our tool also provides the possibility to determine reasonable variable ranges automatically. The idea of the algorithm for some program variable x is to perform a binary search in the interval $[1, k]$, where k is an upper bound for the bit size of x until an element i has been found such that $|\text{MDP}(\tilde{P}, x, i)| = |\text{MDP}(\tilde{P}, x, k)|$. Here, $|\text{MDP}(\tilde{P}, x, i)|$ denotes the number of states in the MDP for $\tilde{P}$ when the bitsize of x is i. For efficiency purposes we implemented a modfied version of this algorithm that starts with bitsize 1 and then increase it to the next 8 bit-border. If the model size changes we decrease by 4 bit to see if the lower size suffices as well, and so on.

5 Implementation and results

The translation described in Section 3 and the heuristics of the previous section have been implemented on the top of our model checker LiQuor [4] and linked to the PRISM model checker. We called the resulting tool Prismela. It runs under the operating system *Microsoft Windows*. Using a graphical user interface (see Fig. 3) the user is able to load a Probmela model, control the translation process regarding variable orderings, variable ranges and start PRISM to build the model. It is also possible to combine user knowledge and automated procedures, for instance when the user already knows the value domain of particular variables or wants to fix the position of certain variables in the variable ordering. Then these variables can be excluded from the heuristics and variable range finding process. Furthermore the user has the option for manual changes on generated PRISM code that can then be exported for furthergoing use in PRISM. The relevant parts of LiQuor (The Probmela-compiler, PASM assembler, the virtual machine that generates the PRISM lanuage model from the assembler code) were linked to Prismela so that the application runs independently from LiQuor.

Our model checker LiQuor [4] uses an intermediate representation of the Probmela program rather than the textual representation of the Probmela program itself. This intermediate representation is the result of a compiling process done by a compiler that was designed to translate Probmela into an assembler like formalism, called probabilistic assembler language (PASM), which is executed on a stack-based virtual processor during the model checking procedure. The virtual machine is connected to a storage module that can save and restore encountered the states of the MDP. The PASM-based approach has several advantages. One of them is that the correctness of the Probmela compiler can be easily be established by checking that the generated PASM code is consistent with the control graph semantics of Probmela. The crucial point for the purposes of this paper is that the generation of the PRISM language model can start from the PASM code rather than the Probmela specification. The generation of the PRISM modules is obtained by realizing the above translation steps for control edges on the level of PASM micro-commands. As a side effect, our translation is not affected by future extensions of Probmela (as long as they yield PASM code with a semantics based on control graphs as above) and is applicable to any other formalism with a PASM-translator.

Figure 4 shows some experimental results of the translation scheme. Among the case studies is one industrial motivated model (UMTS) that involves examining certain rare errors that occur when UMTS phones register to the network provider. This model involves complex storage behaviour in internal buffers of an UMTS end user device and uses almost

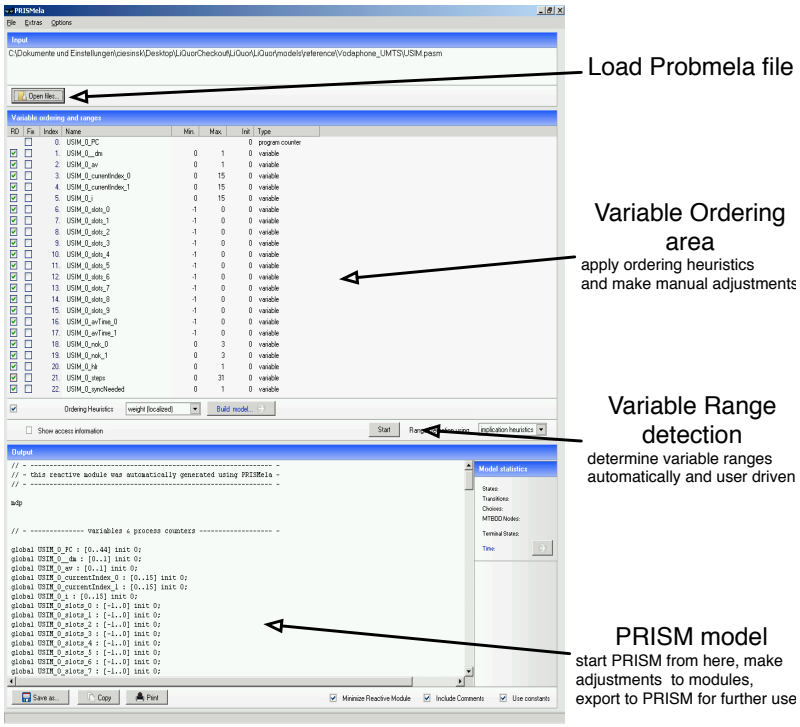

Fig. 3. Graphical user interface and functionality.

every language element of Probmela discussed in this paper. Values given in parantheses are parameters for sizes and other characteristics and are not explained in detail. Larger numbers here indicate larger buffer tables and a larger number of potential entries in these tables, thus resulting in a larger model. Furthermore the table contains results from a randomized variant of the Dining Philosophers [12] (number of processes in parenthesis) and results from a randomized version of the Leader Election protocol [11] (number of processes in parenthesis). The results show that there exist models where one tool experiences great difficulties where the other may succeed rather quick, and vice versa. As expected for smaller models the explicit approach of LiQuor outperforms PRISM's symbolic approach while the state explosion problem is more severe for the explicit approach of LiQuor.

Figure 5 illustrates the efficiency of our translation algorithm. The fanin-heuristic (as well as randomly chosen orderings) leads to very large MTBDDs. The amount of time to build the MTBDDs was always significantly lower when the weight heuristic was applied to calculate a variable ordering.

model	MDP states	MDP transitions	time (LiQuor)	time (PRISM)	MTBBD-nodes
UMTS (10/3/20)	17.952	18.539	$< 1s$	17s	132.895
UMTS (30/5/60)	177.416	186.063	2s	1166s	$1.4 \cdot 10^6$
Din.Phil. (3)	635	2.220	$< 1s$	$< 1s$	2.011
Din.Phil. (6)	411255	$2.8 \cdot 10^6$	$92s$	$< 1s$	9645
Din.Phil. (10)	$2.2 \cdot 10^9$	$26 \cdot 10^9$	–	3s	41.953
Leader El.(3)	1.562	4.413	$< 1s$	$< 1s$	3410
Leader El.(6)	$4.2 \cdot 10^6$	$23 \cdot 10^6$	664s	6s	69.515
Leader El.(10)	$1.9 \cdot 10^{11}$	$1.7 \cdot 10^{12}$	–	–	926.585

Fig. 4. Some results with case studies

6 Conclusion and future work

We presented an approach for the automatic translation of Probmela into the PRISM language. We thus can obtain an MTBDD representation for the Probmela program using PRISM. The translation process presented here is independent of the input language Probmela as it works on control graphs. It is therefore flexible for extensions of the input language and is, in principle, applicable to other modeling languages with a control graph semantics.

We also presented heuristics that serve to optimize the generation of the MTBDD from PRISM programs. These heuristics operate only on PRISM level and can therefore be applied to any PRISM program.

Future work on the presented topics include exhaustive comparisons between the symbolic and explicit model checking approach for probabilistic systems. Further improvements of the translation include language elements that were not covered yet. Probmela (as well as Promela) allows for dynamic creation of processes that would also be a desirable feature for Prismela.

Another target of future work will be the impact of static partial order reduction of Probmela programs on the use with symbolic model checkers.

References

1. R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, July 1999.
2. C. Baier, F. Ciesinski, and M. Größer. Probmela: a modeling language for communicating probabilistic systems. In *Proceeding MEMOCODE*, 2004.
3. Michael Baldamus and Jochen Schröder-Babo. p2b: a translation utility for linking promela and symbolic model checking (tool paper). In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 183–191, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
4. F. Ciesinski and C. Baier. LiQuor: a tool for qualitative and quantitative linear time analysis of reactive systems. In *Proc. QEST*, pages 131–132. IEEE CS Press, 2007.

7 Dining Philosophers,
$3.3 \cdot 10^6$ states, $26 \cdot 10^6$ transitions.
15 PRISM variables (42 bits), 144 PRISM actions.

heuristic	MTBDD nodes	time
weight	9766	0.6s
fanin	51766	6s
random ordering (mean value)	61617	7s

10 Dining Philosophers,
$1.9 \cdot 10^9$ states, $22 \cdot 10^9$ transitions.
21 PRISM variables (51 bits), 202 PRISM actions.

heuristic	MTBDD nodes	time
weight	19684	2,2s
fanin	891604	359s
random ordering (mean value)	356627	171

Leader Election, 7 instances,
$62 \cdot 10^6$ states, $398 \cdot 10^6$ transitions.
28 PRISM variables (56 bits), 112 PRISM actions.

heuristic	MTBDD nodes	time
weight	$1,4 \cdot 10^5$	35s
fanin	$2,4 \cdot 10^6$	818s
random ordering (mean value)	$2,4 \cdot 10^6$	438s

Leader Election, 10 instances,
$194 \cdot 10^9$ states, $17 \cdot 10^11$ transitions.
28 PRISM variables (80 bits), 142 PRISM actions.

heuristic	MTBDD nodes	time
weight	886510	1081s
fanin	—	—
random ordering (mean value)	—	—

UMTS, 10/3/30,
35202 states, 36413 transitions.
23 PRISM variables (57 bits), 136 PRISM actions.

heuristic	MTBDD nodes	time
weight	218662	38s
fanin	233484	64s
random ordering (mean value)	252813	90s

Fig. 5. Influence of variable ordering heuristics on model generation with PRISM.

5. Frank Ciesinski, Christel Baier, Marcus Groesser, and Joachim Klein. Reduction techniques for model checking markov decision processes. *submitted for publication*, 2008.
6. E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *International Workshop on Logic Synthesis*, Tahoe City, 1993.
7. H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming: Special Issue on Probabilistic Techniques for the Design and Analysis of Systems*, 56(1-2):23–67, 2003.
8. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.
9. G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
10. Shin ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 52–57, New York, NY, USA, 1990. ACM Press.
11. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1), 1990.
12. D. Lehmann and M. O. Rabin. On the advantage of free choice: A symmetric and fully distributed solution to the Dining Philosophers problem (extended abstract). In *Proc. Eighth Ann. ACM Symp. on Principles of Programming Languages*, pages 133–138, 1981. A classic paper in the area of randomized distributed algorithms. They show there is no deterministic, deadlock-free, truly distributed and symmetric solution to the Dining Philosophers problem, and describe a simple probabilistic alternative.
13. S. Malik, A.R. Wang, and R.K. Brayton. Logic verification using binary decision diagrams in a logic synthesis environment. In *ICCAD-88: Digest of technical papers*, pages 6–9. IEEE Press, 1988.
14. K. L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.
15. C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design: OBDD-Foundations and Applications*. Springer-Verlag, 1998.
16. D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.
17. PRISM web site. www.prismmodelchecker.org.
18. Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, 1994.
19. R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE /ACM International Conference on CAD*, pages 188–191, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.
20. V.Beaudenon, E. Encrenaz, and S.Taktak. Data decision diagrams for promela systems analysis. *Software Tools and Technology Transfert (accepted for publication)*, 2008.
21. I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Monographs on Discrete Mathematics and Applications. SIAM, 2000.