Translation of Timed Promela to Timed Automata with Discrete Data

Wojciech Nabiałek†
Institute of Computer Science, University of Podlasie
ul. Sienkiewicza 51, 08-110 Siedlce, Poland
wojtek@iis.ap.siedlce.pl

Agata Janowska, Paweł Janowski
Institute of Informatics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
{janowska, janowski}@mimuw.edu.pl

Abstract. The aim of the work is twofold. In order to face the problem of modeling time constraints in Promela, a timed extension of the language is presented. Next, timed Promela is translated to timed automata with discrete data, that is timed automata extended with integer variables. The translation enables verification of Promela specifications via tools accepting timed automata as input, such as VerICS or Uppaal. The translation is illustrated with a well known example of Fischer’s mutual exclusion protocol. Some experimental results are also presented.

Keywords: Promela, Spin, Timed Automata, timed systems, model checking

1. Introduction

Model checking, as a verification method is considered as one of the most interesting and promising. A number of tools (model checkers) was developed. One of the most popular is Spin [9], the first system offering a complete methodology of automatic verification using model checking. Spin accepts specifications of systems being verified written in Promela (Process or Protocol Meta Language). Properties to

†Address for correspondence: Inst. of Comp. Sc., University of Podlasie, ul. Sienkiewicza 51, 08-110 Siedlce, Poland

*The work is partly supported by the Polish grant No. 3T11C01128.
be checked can be specified as invariants, linear temporal logic (LTL) formulas or so-called never claims that represent Büchi automata (to reason about infinite executions).

The verification with Spin consists in automatically checking all the properties against the behavioral part of the system by an exhaustive search. This task is done in a very efficient way with use of techniques such as partial order reduction, bit state hashing and state compression.

A number of verification systems follows the method of a direct state representation. The most popular ones are Kronos [6] and Uppaal [3], which differ from Spin in fact that they can also deal with timed systems. There is another approach to the model checking problem based on a symbolic representation of states. To the category of tools based on an indirect state space representation belongs the verification platform VerICS [7] used in experiments in this paper but also e.g. Uppaal2k [11] or SMV [5].

What can be seen as a disadvantage of Promela is a lack of possibility to express time constraints which play an important role in many modern protocols. There were two attempts to extend the language — Real Time Promela proposed in [12] and Discrete Time Promela [4]. In both cases special verification tools have been constructed, namely RTSpin and DTSpin. We propose a different approach. In the paper new timed extension of Promela is introduced as well as its translation to timed automata with discrete data (TADD for short) [10], which is an input formalism of VerICS and other tools such as already mentioned Uppaal or HyTech [8]. The translation makes possible verification of Promela specifications (timed and untimed) with couples of new tools and comparison of efficiency between Spin and other model checkers.

The paper begins with a presentation of Promela and its timed extension. Next, timed automata with discrete data are introduced. Then, the translation of Promela to a set of TADDs is given illustrated with an example of the mutual exclusion protocol. Finally, experimental results are presented. Conclusions and future works sum up the paper.

2. Promela and Timed Promela

Promela is loosely based on Dijkstra guarded command language. A system is described as a set of processes communicating via shared variables and channels supporting rendez-vous and buffered message passing. A full presentation of Promela, a complex language, is out of the scope of the paper. We present only a brief overview of its substantial subset.

2.1. Promela

Promela constructs can be divided into two categories: behavioral and verification ones. The behavioral part is used for representing the actual behavior of the systems. The systems are modeled as a sets of instances of communicating process. A process is defined as a sequence of possibly labeled instructions preceded by the declarative part. Initially, only one process is executed: a process init that must be declared explicitly in every specification. New processes are instantiated using the run statement.¹

Data structures. Promela offers a number of basic data-types, such as: bit, bool, byte, short or int. Arrays are also supported. The index of an array can be any expression that determines a unique

¹We present the translation under the assumption that run statement is used only in the body of the init process.
integer value. A Promela typedef definition can be used to introduce a new name for a list of data objects of predefined or already defined types. Message channels are used to model the transfer of data between processes. With each channel it is associated its size that is a maximal number of elements that can be stored in the channel. The special channels of the size 0 are used to model the synchronous communication (rendez-vous).

**Basic statements.** Instructions that can modify the system state are assignments and instructions for sending (receiving) messages to (from) channels. Boolean expressions define tests over variables and contents of channels. In particular, boolean expressions include tests for fullness and emptiness of a channel. Promela contains also the predefined Boolean expressions skip (which is basically equal to true). There are two basic operations on channels. The statement ch! expr denotes sending the value of the expression expr to the channel ch. The value is appended to the end of the channel. The statement ch? y represents getting the message from the beginning of the channel and assigning it to the variable y. In case of channels with the size 0 the operations of sending and receiving are executed synchronously with the effect of the assignment of the value of the expression expr to the variable y.

**Control flow.** The control flow instructions such us selection statements if...fi; and iterative statements do...od; with usual meaning of imperative languages are also present in the form:

```plaintext
if
    :: <condition1> -> <instructions1>
    :: <condition2> -> <instructions2>
...
fi
```
(alternative notation for a statement of one branch is <condition> -> <instructions>). Branches in selections and loops are non deterministic. If all guards are not satisfied, then the process is blocked until one of them can be selected.

There are two additional statements that can be used as guards: the timeout statement and the else statement. The timeout branches can be executed only if no other instruction of the whole system is enabled. The else statement can be executed only if all other guards of the selection are not satisfied. The break statement is a way to end the loop. As usual the goto statement moves the control to any named label in the process body.

Promela atomic construction is used to implement atomic sequences of instructions. By prefixing a sequence of statements with the atomic keyword the user can indicate that the sequence is to be executed as a single unit, non-interleaved with any other processes.

The main-stmt unless escape-stmt construction introduces a kind of dynamic priorities. The escape-stmt has higher priority than main-stmt, which means that main-stmt can be executed only if escape-stmt is not executable, otherwise escape-stmt is executed.

**Never claims.** Never claim is a special type of a process that, if present, is instantiated once. Never claims are used to detect behaviors that are considered undesirable or illegal. The verifier will complain if there is an execution that ends in a state in which the never claim terminates.
2.2. Mutual exclusion protocol example

The example protocol is very simple (and incorrect). Before entering the critical section each process checks whether any other process is in it. The variable \( is_i \) (initially equal to 0) takes the value 1 when the \( i \)-th process enters the section and is set to 0 upon leaving it. The Promela description of the process \( P_1 \) is presented below (the number of all processes is 4).

\[
\text{proctype } P_1() \{ \\
\text{do :: true } \rightarrow \text{ do :: (} is_2 \neq 1 \text{) } \&\& \text{ (} is_3 \neq 1 \text{) } \&\& \text{ (} is_4 \neq 1 \text{) } \rightarrow \text{ break; od; } \\
is_1 = 1; \\
atCS_1: \\
is_1 = 0; \\
\text{od } 
\}
\]

For this protocol we want to check if all the processes can stay in the critical section at the same time. Such property can be expressed using the following never claim:

\[
\text{never } \{ \\
\text{do :: } P_1@atCS_1 \&\& P_2@atCS_2 \&\& P_3@atCS_3 \&\& P_4@atCS_4 \rightarrow \text{ goto end; } \\
\text{:: else } \rightarrow \text{ skip; } \\
\text{od; } \\
\text{end: } 
\}
\]

2.3. Timed Promela

**Discrete Time and Real Time Promela.** In Discrete Time Promela [4] a new data type called \texttt{timer} is introduced to represent discrete-time countdown timers. Three new statements operating on timers are: \texttt{set(tmr, val)} that activates the timer \( \texttt{tmr} \) with \( \texttt{val} \) as its initial value, \texttt{reset(tmr)} that deactivates the timer and \texttt{expire(tmr)} that tests whether the timer value is 0. Based on these statements some macros are defined. For example \texttt{delay(tmr, val)} models delay of \( \texttt{val} \) units of time and is defined as the sequence \texttt{set(tmr, val)}; \texttt{expire(tmr)}. The \texttt{udelay(tmr, val)} macro represents unbounded delay and \texttt{bdelay(tmr, val)} — nondeterministic bounded delay. The additional \texttt{tick} operation, which is used internally by DTSpin to represent time progress, nondeterministically decreases the values of all active timers by 1.

Real Time Promela [12] is a real time extension which introduces global \texttt{clock} variables. The value of a clock can be set to 0 with the \texttt{reset{clk}}. Each Promela statement can be expanded with \texttt{when{constr}} which defines timed conditions on a clock which must be fulfilled to execute the statement. Similarly to our approach the mathematical foundations of Real Time Promela is the formalism of Timed Automata.

**Timed Promela.** In our extension no timers, nor clocks are used. Instead, one can impose time bounds on the execution of a statement with two new statements \texttt{wait} and \texttt{timeout} as follows:
if
  :: <condition1> -> <instructions1>
  :: [wait <DL>] <condition2> -> <instructions2>
  ...
  :: [timeout <DU>] --> <instructions3>
fi

On the one hand moment of execution of a branch can be delayed of at least DL units of time with [wait DL] statement (a lower time bound), on the other hand an upper timed bound can be defined by [timeout DU] statements. All branches of a selection have the same upper time bound. It means that instructions associated to timeout are executed exactly after DU units of time, but only if no other branch is enabled. Otherwise the execution of an enabled branch is enforced.

<table>
<thead>
<tr>
<th>Real Time Promela</th>
<th>Discrete Time Promela</th>
<th>Timed Promela</th>
</tr>
</thead>
<tbody>
<tr>
<td>proctype P(byte id) { do :: reset{y[id]} X=0; } when{y[id] &lt; deltaB} reset{y[id]} X=id+1; atomic{ when{y[id] &gt; deltaC} X=id+1; in_crit++ ; } od }</td>
<td>proctype P(byte id) { timer y, y1; do :: udelay(y); X=0; } when{ y[id] &lt; deltaB} reset{y[id]} X=id+1; atomic{ bdelay(y,deltaB,y1) X=id+1; } atomic{ delay(y, deltaC); udelay(y) X=id+1; in_crit++ ; } atomic{ X=id+1; in_crit++; } atomic{ X=0; in_crit--; } od }</td>
<td>proctype P(byte id){ do :: X=0; } if :: true X=id+1; :: [timeout deltaB] --&gt; skip; fi; [wait deltaC] X=id+1; in_crit++; atomic { inCrit--; X=0; } od }</td>
</tr>
</tbody>
</table>

Figure 1. Fischer’s Mutual Exclusion Protocol.

Fig. 1 shows a process definition of Fischer’s Mutual Exclusion protocol [1] written in all presented time extensions of Promela. The reader itself can compare the readability of the specifications. Experimental results for this case are given in Section 5.

3. Timed Automata with Discrete Data

In this section we introduce timed automata with discrete data based on standard Alur-Dill timed automata [2] extended with integer variables. We present here only main notions about TADDDs. The details can be found in [10].

Let \( V \) be a finite set of integer variables and \( \text{Expr}(V) \) be the set arithmetic expressions over \( V \), \( B\text{Expr}(V) \) — the set of boolean expressions over \( V \), and \( \text{Act}(V) \) — the set of all the actions over \( V \),
that is sequences of assignments of the form $y := expr$, where $y \in V$ and $expr \in \text{Expr}(V)$. Let $X$ be a finite set of real variables, called clocks. The set $\Psi(X)$ of all the clock constraints over $X$ is the conjunction of the constraints of the form $x \sim c$, where $x \in X, c \in \mathbb{N}$, and $\sim \in \{\leq, <, =, >, \geq\}$.\(^2\)

A Timed automaton with discrete data (TADD for short) is a tuple $A = (\Sigma, Q, q^0, V, X, T, I)$, where

- $\Sigma$ is a finite set of labels,
- $Q$ is a finite set of locations,
- $q^0$ is the initial location,
- $V$ is a finite set of integer variables,
- $X$ is a finite set of clocks,
- $T \subseteq Q \times \Sigma \times B\text{Expr}(V) \times \Psi(X) \times \{\text{true}, \text{false}\} \times \text{Act}(V) \times 2^X \times Q$ is a transition relation,
- $I : Q \longrightarrow \Psi(X)$ is an invariant function,
- $C : Q \longrightarrow \{\text{true}, \text{false}\}$ is an commitment function.

An element $t = (q, l, \phi, \psi, u, \alpha, Y, q')$ of $T$ denotes a transition from the location $q$ to the location $q'$, where $l$ is the label of transition $t$, $\phi$ and $\psi$ define the enabling conditions (the guard and the clock constraint) for $t$, $\alpha$ is the action to be executed, $Y$ is the set of clocks to reset (reset set), and $u$ is the urgency attribute – when it is set, the transition $t$ has to be executed as soon as it is enabled ($t$ is enabled if the automaton is in the location $q$ and the guard $\phi$ evaluates to true). We assume that clock constraints of urgent transitions are equal to true. The invariant function assigns to each location $q \in Q$ a clock constraint defining the condition under which the automaton can stay in $q$. The commitment function $C$ marks some locations as committed. If a set of TADDs is running parallel, then transitions going out of committed locations have priority over the other transitions. We assume that $C(q^0) = 0$.

A state of an TADD is a tuple $(q, v, \tau)$, where $q \in Q, v \in V^\mathbb{Z}$, and $\tau \in X_{\mathbb{R}+}$. Initially, all the variables have values defined by a given initial valuation $v^0$ and all the clocks are set to 0. At a state $s = (q, v, \tau)$ the system can either:

- execute an enabled transition $t$ and move to the state $s' = (q', v', \tau')$, where $q'$ is the target of $t$, the variable valuation is changed according to the action of $t$ and the clocks from the reset set of $t$ are set to 0, or
- let time $\delta$ pass and move to the state $(q, v, \tau + \delta)$, as long as no urgent transition is enabled and $\tau + \delta$ satisfies $I(q)$.

\(^2\)\(\mathbb{Z}\) denotes the set of integer numbers, $\mathbb{N}$ — the set of non-negative integer numbers and $\mathbb{R}^+_{\text{IR}}$ — the set of non-negative real numbers.
Parallel Composition of TADDS. We assume that a system to be verified is described as a set of automata running parallel. Automata communicate with each other via shared variables. It is required that the transitions with a shared label have to be executed synchronously by all the automata containing this label. Transitions with unique (local) labels are interleaved. To obtain clear semantics of it is necessary to fix the order of actions in case of synchronous transitions. The transition whose action should be taken first is marked with ! and it is called an output transition. Transitions whose actions should be taken next are marked with ? and they are called an input transitions. We assume that input transitions do not update shared variables.\(^3\)

Let \( \{A_1, \ldots, A_n\} \) be a set of timed automata and \( \Sigma(l) = \{1 \leq i \leq n \mid l \in \Sigma_i\} \) denote the set of the numbers of the automata containing the label \( l \). By \( \epsilon \) we denote an empty string which concatenated with a character \( l \) gives \( l \). Additionally, by \( \{a_j\}_{j \in Z} \) we denote the sequence of the actions \( a_j \), where \( j \in Z \subseteq \{1, \ldots, n\} \), such that the actions are arranged according to the numerical order of their indices.

Let \( A_i = (\Sigma_i \times \{!?, \epsilon\}, Q_i, q^0_i, V_i, X_i, E_i, I_i) \) for \( i = 1, 2, \ldots, n \) be timed automata such that \( X_i \cap X_j = \emptyset \) for all \( i \neq j \). A parallel composition of the above automata (denoted as \( A_1 \parallel A_2 \parallel \ldots \parallel A_n \)) is the timed automaton \( A = (\Sigma, Q, q^0, V, X, E, I) \), where:

- \( \Sigma = \bigcup_{i=1}^{n} \Sigma_i \),
- \( Q = \prod_{i=1}^{n} Q_i \),
- \( q^0 = (q^0_1, \ldots, q^0_n) \),
- \( V = \bigcup_{i=1}^{n} V_i \),
- \( X = \bigcup_{i=1}^{n} X_i \),

transition relation \( E \) is defined as follows: \( ((q_1, \ldots, q_n), l, \phi, \psi, u, \alpha, \rho, (q'_1, \ldots, q'_n)) \in E \) iff

- for all \( i \in \Sigma(l) \) there exists \( (q_i, l, \phi_i, \psi_i, u_i, \alpha_i, \rho_i, q'_i) \in E_i \), such that \( l_i \simeq l \) and there exists \( j \in \Sigma(l) \) such that \( l_j = l \) and \( \phi = \bigwedge_{i \in \Sigma(l)} \phi_i \), \( \psi = \bigwedge_{i \in \Sigma(l)} \psi_i \), \( u = \bigwedge_{i \in \Sigma(l)} u_i \), \( a = \{a_j\}_{j \in \Sigma(l)} \), \( \rho = \bigcup_{i \in \Sigma(l)} \rho_i \) and for all \( i \in \{1, \ldots, n\} \setminus \Sigma(l) \), \( q'_i = q_i \), or
- there exists \( 1 \leq j \leq n \) such that \( \Sigma(l) = \{j\} \), \( (q_j, l, \phi, \psi, u, \alpha, \rho, q'_j) \in E_j \) and for all \( i \in \{1, \ldots, n\} \setminus \{j\} \), \( q'_i = q_i \).

\( I(q_1, \ldots, q_n) = \bigwedge_{i=1}^{n} I_i(q_i) \) and \( C(q_1, \ldots, q_n) = \bigvee_{i=1}^{n} C_i(q_i) \).

4. From Promela to Timed Automata with Discrete Data

This section describes the method of the translation from the Promela specification into the set of timed automata with discrete data. As Promela is a quite complex language we cover its substantial subset only. The most important restrictions concern embedded C extensions and dynamic process creation. Also, we perform the translation under the assumption that the unless and timeout (original) statements are not used in the specification. However, we describe how to handle these statements in the discussion at the end of this Section. We require also that in case of rendez-vous any pair of processes uses a different channel.

\(^3\)These variables are used by more than one process.
The translation is performed in three stages. The first one consists in transformation of control flow of each Promela process into an automata structure. The next one concerns representation of Promela data structures and operations on them. Finally, we define a set of TADDs corresponding to all instances of Promela processes.

The aim of the first two stages is to define for each instance of a Promela process a tuple \( \text{proc} = (\text{pid}, \text{lvars}, \text{lstates}, \text{initial}, \text{trans}) \), where \( \text{pid} \) is an integer that uniquely identifies the process, \( \text{lvars} \) is a set of variables local to the process, \( \text{lstates} \) is a finite set of states, and \( \text{initial} \) is the initial state. A set \( \text{trans} \) of transitions is composed of tuples of the form \( \text{tran} = (\text{tid}, \text{source}, \text{target}, \text{cond}, \text{effect}, \text{ltime}, \text{utime}, \text{rv}) \), where \( \text{tid} \) is a transition identifier, \( \text{source} \) and \( \text{target} \) are states, \( \text{cond} \) is a boolean condition, \( \text{effect} \) is a sequence of basic statements (that is assignments, send or receive statements), \( \text{ltime} \) and \( \text{utime} \) are integers (\( \text{utime} \) can be also defined as \( \infty \)) representing lower and upper time bounds of a transition execution, and \( \text{rv} \) is either of the form \( \text{ch}! \) or \( \text{ch}? \), if the effect of the transition is synchronous sending or receiving using a special 0-size channel \( \text{ch} \), or \( \text{null} \) otherwise. Finally, a state \( \text{state} = (\text{sid}, \text{atomic}) \) is defined by an identifier \( \text{sid} \) and a boolean value \( \text{atomic} \).

4.1. Structural transformation

The set of basic Promela statements is small. It contains: assignments, send and receive statements. The other language constructions such as selections, repetitions, and jumps serve only to define possible control flows. The transformation method is inductive. Let \( \text{Tr(stmt)} \) denote an automaton built for a statement \( \text{stmt} \). The procedure starts with a block (a sequence of statements) representing the behavior of a whole process and operates in a top-down fashion upto basic statements. As the translation is very technical we do not present it systematically in full details, but we focus on its crucial aspects. The rest of details concerning definitions of particular elements of the \( \text{proc} \) structure should be clear from the context.

**Sequences of statements.** For a sequence of \( n \) statements we produce a sequence of \( n + 1 \) states with \( n \) transitions between them — each transition corresponds to one statement (Fig. 2 a). If a sequence of statements is grouped with \text{atomic} keyword we denote each inner state \( s \) as committed (Fig. 2 b), that is \( s.\text{atomic} = \text{true} \), otherwise \( s.\text{atomic} = \text{false} \).

\[ \text{a) Tr(instr1;...;instrn)} \]

\[ \text{b) Tr(atomic(instr1;...;instrn;))} \]

Figure 2. Translation of a sequence of statements (C denotes committed states).

\[^4\text{We refer to the elements of a tuple with a dot notation.}\]
Selections and repetitions. A conditional statement (if) is represented as a set of transitions — one for each branch of a selection as depicted at Fig. 3. If in an $i$th branch of a selection there is a \([\text{wait} \, D_i]\) statement, then for the transition $\text{tran}$ corresponding to the branch, $\text{trans}.\text{lt} = D_i$, otherwise $\text{trans}.\text{lt} = 0$. Also, if in a conditional statement \([\text{timeout} \, DT]\) is used, then for each transition $\text{tran}$ corresponding to any of the branches $\text{trans}.\text{ut} = DT$, otherwise $\text{trans}.\text{ut} = \infty$. Note, that else and timeout statements are mutually exclusive. A repetition statement (do) is treated in a very similar way (Fig. 4).

4.2. Translation of basic statements.

Data representation. All scalar Promela variables are mapped directly onto TADD variables (with the same names), whereas arrays and channels are represented by a set of TADD variables.

Each Promela channel $ch$ is represented as a tuple $(\text{chid}, \text{nslots})$, where $\text{chid}$ is a unique identifier and $\text{nslots}$ is the size of the channel. Similarly, an array $a$ is represented by $(\text{aid}, \text{nslots})$. Let $\text{gvars}$ be the set of all scalar Promela variables, $\text{arrs}$ be the set of all arrays and $\text{chans}$ be the set of all channels (for simplicity we assume that all arrays and channels are global).
There are two types of channels: asynchronous (with the size greater than 0) and synchronous (with size 0). Each asynchronous channel \(ch\) is modeled by \(\text{size}(ch)\) TADD variables. Let \(V_{ch}\) be the set \(\{ch_{\text{length}}, ch_{1}, \ldots, ch_{\text{nslots}(ch)}\}\). The variable \(ch_{\text{length}}\) represents the number of elements stored in the channel and \(ch_{1}, \ldots, ch_{\text{nslots}(ch)}\) models the channel contents. Similarly, an array \(a\) is modeled by the set \(V_a = \{a_{1}, \ldots, a_{\text{nslots}(a)}\}\). To each synchronous channel \(ch\) corresponds a single variable \(ch_{\text{val}}\).

**Assignments.** An assignment of an expression to a (Promela) variable of the form \(y = \text{expr}\) is represented with a single transition \(\text{tran}\), which effect is an assignment of the expression to the corresponding TADD variable. If no array is used in the expression, then \(\text{tran.effect} \equiv y := \text{expr}\). An operation on an array \(a\) is represented as a set of \(a_{\text{nslots}}\) transitions. Each transition corresponds to one of the possible values of the array index. Fig. 5 presents translation of statements \(a[i] = \text{expr}\) and \(y = a[i]\).

**Operation on channels.** An operation on asynchronous channel \(ch\): \(ch!\text{expr}\) is translated to \(ch_{\text{nslots}}\) transitions of the form depicted in Fig. 6. Puting a piece of data into the channel corresponds to assigning it to the first free variable representing the channel contents. If the channel is full a process has to wait for the free space. Getting data from a channel is just reading from the variable \(ch_{1}\) (provided that the channel is not empty) and shifting all the remaining elements left. If, during the verification, we want to check for instance whether a process tries to read from an empty channel, then an additional state is created with incoming transition having the guard \(\text{cond}\) of the form \(ch_{\text{len}} = 0\).

Operations on a synchronous channel \(ch\) are represented as transitions to be executed synchronously by TADDs representing two communicating processes. For a transition \(\text{tran}\) corresponding to a sending operation \(ch!\text{expr}\): \(\text{tran.rv} \equiv ch!\) and \(\text{tran.effect} \equiv (ch_{\text{val}} := \text{expr})\). For a transition \(\text{tran}'\) corresponding to a receiving operation \(ch?y\): \(\text{tran'.rv} = ch?\) and \(\text{tran'.effect} = (y := ch_{\text{val}})\).

![Figure 5. Translation of operations on an array (n = a.nslots).](image-url)
Each Promela boolean expression be xpr has a counterpart either of a single transition with \( \text{cond} \) \( = \) be xpr (when no array is used), or of a set of transitions (analogously to the other operations on arrays).

### 4.3. TADD construction

Given a Promela specification \( \text{spec} = (gvars, arrs, chans, procs) \) obtained as a result of the previous transformations we construct a TADD for each process from the set \( \text{procs} \).

#### Translation of a Promela transition.

For a transition \( \text{tran} = (\text{tid}, \text{source}, \text{target}, \text{cond}, \text{effect}, \text{ltime}, \text{utime}, \text{rv}) \) of a process \( \text{proc} \) a TADD transition \( \text{Tr}(\text{tran}) = (q, l, \phi, \psi, u, \alpha, Y, q') \) is defined as follows:

- \( q = \text{tran.source} \),
- if \( \text{tran.rv} \neq \text{null} \), then \( l = \text{tran.rv} \), otherwise \( l \) is a string \( \text{tr}((\text{tran.tid}) \text{of} \{\text{proc.pid}\}) \),
- \( \phi = \text{tran.cond} \),
- \( \psi = (x_i \geq \text{tran.ltime}) \),
- \( u = \text{false} \),
- \( \alpha = \text{tran.effect} \),
- \( Y = \{x_i\} \) if there is a transition \( \text{tran}' \) such that \( \text{tran}'.source = \text{tran}.target \) and \( (\text{tran}'.utime \neq \infty \text{ or } \text{tran}'.ltime \neq 0) \), otherwise \( Y = \emptyset \),
- \( q' = \text{tran.target} \).
In case of rendez-vous the label of a transition is meaningful, otherwise it has only informational meaning.

**Translation of a Promela process.** For a Promela process \( P = (\text{pid}, lvars, lstates, initial, trans) \), we define a TADD \( A_i = (\Sigma_i, Q_i, q_i^0, V_i, X_i, E_i, I_i) \), where \( i = \text{pid} \) and:

- \( \Sigma_i \) is composed of labels of the form \( l = tr\langle\text{tid}\rangle \) of \( \langle\text{proc.pid}\rangle \), for each transition identifier \( \text{tid} \), and \( \text{ch!} \) or \( \text{ch?} \) for each synchronous channel \( \text{ch} \) used by the process \( \text{proc} \),
- \( Q_i = \text{proc.lstates} \cup \{\text{error}_i\} \),
- \( q_i^0 = \text{proc.initial} \),
- \( V_i = \text{proc.lvars} \cup \text{spec.gvars} \cup \bigcup_{\text{ch} \in \text{spec.chans}} V_{\text{ch}} \cup \bigcup_{a \in \text{spec.arrs}} V_{a} \),
- \( X = \{x_i\} \), if there exists a transition \( \text{tran} \in \text{proc.trans} \) such that \( \text{tran.ltime} \neq 0 \) or \( \text{tran.utime} \neq \infty \), otherwise \( X = \emptyset \),
- \( E_i = \bigcup_{\text{tran} \in \text{proc.trans}} \text{Tr}(\text{tran}) \),
- for \( q \in Q_i \) the invariant function \( I_i(q) \) is of the form \( x_i \leq \text{tran.utime} \), if there is a transition \( \text{tran} \in \text{proc.trans} \) such that \( q = \text{tran.source} \) and \( \text{tran.utime} \neq \infty \), otherwise \( I(q) = \text{true} \).

**Never claim.** The translation of a never claim is slightly different. If a never claim is solely based on states mentioned in the specification, like in example of Section 2.2, it can be easily translated into a temporal formula. Otherwise an additional TADD is build as a part of specification in a similar way like for regular processes.

**4.4. Example**

Fig. 7 presents the TADD obtained for process \( P_0 \) of the protocol presented in Fig. 1. For clarity we skip the transition labels.

![Diagram](image_url)

**Figure 7.** TADD for a process \( P_0 \) of Fischer’s protocol.

\(^{5}\text{Note, that values of utime are equal for all transitions outgoing of the same state.} \)
4.5. Discussion

Giving here a formal proof of the correctness of the translation is not possible due to space limits. However, we would like to give some informal justification (sketch of the proof) for case of untimed specifications. First, note that the structure of a TADD $A$ for an instance of Promela process corresponds strictly to its automata structure $P$ given in the semantics definition of chapter 7 of [9]. It means that for each Promela process the set of transitions is exactly the same in $A$ and $P$. Let $S$ denote the set of automata representing processes of a Promela specification and $S'$ be a set of TADDs constructed for the specification. Then, it is not difficult to see that due to the definition of guards of transitions which is basically the same for each $A$ and $P$, the sets of enabled transitions of the systems $S$ and $S'$ are the same. Next, the careful analysis of the Promela semantics engine, which defines which transitions can be chosen to fire, taking into account atomic statements and rendez-vous communication and the semantics of the execution of the set of TADDs with committed locations and transitions synchronizing on the same label, gives that the sets of executable transitions of the systems $S$ and $S'$ are equal. Finally, it should be noticed that, if the systems $S$ and $S'$ are in corresponding states (values of corresponding data structures are equal as well as actual processes states and automata locations), then after executing the corresponding transitions, their states are changed in the same way.

The unless statement is a natural way to model exceptions in modern programming languages (e.g. Java), but it causes problems during the translation, if a rendez-vous communication is used in an escape statement. To handle this statement an alternative definition of choosing a transition to fire is needed, which takes into accounts priorities of transitions corresponding to escape statements. This can be easily done, but requires changes in a verification engine of a tool. We decided to skip this feature and present the translation giving an output which can accepted with any timed automata model checker.

5. Experimental results

Below we present some experimental results. We compare amounts of time and memory needed to complete verification of two cases with Spin (Promela specifications) and VerICS (sets of TADDs obtained from the Promela specifications). Tests were performed on the PC equipped with processor Intel Celeron M 1.73GHz with 512MB of RAM and Slackware 11.0 operating system.

Tables 1 and 2 concern the example presented in Section 2.2. We check two types of properties. The property $\alpha$ states that all the processes are never in the critical section at the same time and property $\beta$ states that any two processes are never in the critical section at the same time. It appears that all the properties are violated.

Two tests were performed using Spin with (results denoted with *) and without partial order reductions. Without partial order reductions Spin can verify only up to 6 processes and then the memory consumption is too high. Applying partial order reductions enables verification of 7 processes and again system goes out of memory. With VerICS it is possible to test the protocol running up to 11 processes with relatively small memory usage.

Table 2 presents results for the other property. In this case we present additionally usage of resources in each of two stages of Spin verification (T1 and M1 concerns the compilation while T2 and M2 — the proper verification). Spin is able to test up to 141 concurrent processes. VerICS could test more than 160 processes.
Table 1. Experimental results for the example of section 2.2 and the property $\alpha$.

<table>
<thead>
<tr>
<th># proc.</th>
<th>Spin mem</th>
<th>Spin cpu</th>
<th>Verics mem</th>
<th>Verics cpu</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>34.04</td>
<td>0.06</td>
<td>3.99</td>
<td>0.15</td>
</tr>
<tr>
<td>4</td>
<td>34.24</td>
<td>0.07</td>
<td>4.61</td>
<td>0.34</td>
</tr>
<tr>
<td>5</td>
<td>40.76</td>
<td>0.72</td>
<td>5.81</td>
<td>2.18</td>
</tr>
<tr>
<td>6</td>
<td>151.94</td>
<td>14.71</td>
<td>7.27</td>
<td>2.98</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>55.89</td>
<td>1.99</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>34.29</td>
<td>70.58</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>84.53</td>
<td>218.49</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>290.98</td>
<td>1245.00</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>318.71</td>
<td>2322.86</td>
</tr>
</tbody>
</table>

Table 2. Experimental results for the example of section 2.2 and the property $\beta$.

<table>
<thead>
<tr>
<th># proc.</th>
<th>Spin M1</th>
<th>Spin M2</th>
<th>Spin mem</th>
<th>Spin cpu</th>
<th>Verics mem</th>
<th>Verics cpu</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.24</td>
<td>15.25</td>
<td>36.97</td>
<td>0.63</td>
<td>1.73</td>
<td>0.37</td>
</tr>
<tr>
<td>50</td>
<td>5.25</td>
<td>163.66</td>
<td>38.81</td>
<td>2.52</td>
<td>13.91</td>
<td>0.83</td>
</tr>
<tr>
<td>100</td>
<td>20.68</td>
<td>607.27</td>
<td>42.91</td>
<td>48.39</td>
<td>67.33</td>
<td>3.82</td>
</tr>
<tr>
<td>141</td>
<td>40.93</td>
<td>1151.77</td>
<td>48.23</td>
<td>202.83</td>
<td>224.23</td>
<td>8.81</td>
</tr>
<tr>
<td>150</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>160</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1. Experimental results for the example of section 2.2 and the property $\alpha$.  
Table 2. Experimental results for the example of section 2.2 and the property $\beta$.  

W. Nabialek et al. / Translation of Timed Promela to Timed Automata with Discrete Data
Table 3. Experimental results for Fischer’s mutual exclusion protocol and the property $\alpha$.

<table>
<thead>
<tr>
<th># proc.</th>
<th>Spin</th>
<th>Verics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RTSpin</td>
<td>DTSpin</td>
</tr>
<tr>
<td></td>
<td>mem</td>
<td>cpu</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 4. Experimental results for Fischer’s mutual exclusion protocol and the property $\beta$.

<table>
<thead>
<tr>
<th># proc.</th>
<th>Spin</th>
<th>Verics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RTSpin</td>
<td>DTSpin</td>
</tr>
<tr>
<td></td>
<td>mem</td>
<td>cpu</td>
</tr>
<tr>
<td>8</td>
<td>34.21</td>
<td>5.4</td>
</tr>
<tr>
<td>80</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>100</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>130</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>135</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Tables 3 and 4 shows experimental results for Fischer’s mutual exclusion protocol (Sections 2.3 and 4.4) for $\delta B = 2$ and $\delta C = 1$. In these cases the protocol was verified using latest distributions of RTSpin, DTSpin and VerICS.

6. Conclusions and future works

We have presented an extention of Promela with timed constraints and the translation of extended Promela to timed automata with discrete data, which can be viewed as a semantics definition for timed Promela. Experimental results show that the efficiency of Promela verification via the translation to TADD is good (at least comparable with verification using Spin). Our future research will focus on extending the efficiency of the translation by incorporating some optimisation methods such as static code analysis.

References


