Translation of Intermediate Language to Timed Automata with Discrete Data*

Agata Janowska[1], Paweł Janowski[1], and Dobiesław Wróblewski[2]

[1] Institute of Informatics, University of Warsaw, Banacha 2, 02-097 Warsaw, Poland
[2] Institute of Computer Science, Polish Academy of Science, Ordona 21, 01-237 Warsaw, Poland
{janowska,janowski}@mimuw.edu.pl, wrobldob@ipipan.waw.pl

Abstract. The aim of this work is to describe the translation from Intermediate Language, one of the input formalisms of the model checking platform VerICS, to timed automata with discrete data and to compare it with the translation to classical timed automata. The paper presents syntax and semantics of both formalisms, the translation rules as well as a simple example.

1 Introduction

A few years ago the first release of the model checking verification platform VerICS [5] was developed. The tool was designed to accept a number of input formalisms which are translated to the common format called the Intermediate Language. Then, for the verification purposes, a system represented as an Intermediate Language program is further translated to either a set of timed automata, each of which represents a component of the system, or to a global (product) timed automaton. The automata obtained in this way are passed to the verification modules. As the VerICS is a subject of the dynamic development in its recent version a new formalism was added, namely timed automata with discrete data, which are timed automata enriched with additional integer variables. Such automata are a common input formalism of other popular model checkers for timed systems (e.g. UppAal [8] and HyTech [7]). In order to introduce timed automata with discrete data to VerICS the new translation from the Intermediate Language is needed and this is a subject of this paper. We would like also to present a comparison with the previous translation. To this end, a few cases described in Intermediate Language is verified via translations to classical timed automata and to extended ones.

The paper is organized as follows. In Section 2 we introduce preliminary notions. Section 3 and 4 contain the syntax and semantics of Intermediate Language and extended timed automata, respectively. Section 5 describes the translation rules between these formalisms. In Section 6 experimental results are presented. The last section shortly concludes the paper.

2 Preliminaries

Variables and buffers. Let V be a finite set of integer variables, and B be a finite set of buffers. A buffer is a possibly empty sequence of integer values. Let $size : B \to \mathbb{N}$ be

* The authors acknowledge a partial support from the Polish grant No. 3T11C01128.

the function that assigns the maximal size (the maximal number of elements) with each buffer. The set $Exp(V)$ of all the *arithmetic expressions* over V is defined by the following grammar:

$$exp ::= m \mid y \mid exp \ \oplus \ exp \mid -exp \mid (exp)$$

where $m \in \mathbb{Z}$, $y \in V$, and $\oplus \in \{-, +, *, /\}$.[3] The set $BExp(V, B)$ of all the *boolean expressions* over V and B is defined inductively as follows:

$$bexp ::= true \mid exp \ \sim \ exp \mid empty(b) \mid bexp \ \ and \ \ bexp \mid bexp \ or \ bexp \mid not \ bexp \mid (bexp)$$

where $exp \in Exp(V)$, $b \in B$, and $\sim \in \{=, \neq, <, >, \leq, \geq\}$. The set $BExp(V)$ is defined in the same fashion (without expressions of the form $empty(b)$). The set $Act(V, B)$ of all the *actions* over V and B is defined as follows:

$$act ::= \ \varepsilon \mid y := exp \mid get(b, y) \mid put(b, exp) \mid act; \ act$$

where ε denotes an empty sequence, $y \in V$, $b \in B$ and $exp \in Exp(V)$. The set $Act(V)$ of all the actions over V is defined in the similar way (without operations *get* and *put*).

Variable and Buffer Valuation. Let $\mathbb{Z}^*$ be a set of all the sequences of integer values. An empty sequence is denoted by ε. We use Ω to denote $\mathbb{Z} \cup \mathbb{Z}^*$. A *variable and buffer valuation* is a total mapping $v : V \cup B \to \Omega$ such that $v(V) \subseteq \mathbb{Z}$ and $v(B) \subseteq \mathbb{Z}^*$. We extend this mapping to expressions in the usual way. A valuation v^0 such that $v^0(b) = \varepsilon$ for all $b \in B$ (all buffers are empty) is called an *initial valuation*. Satisfiability of a boolean expression $\beta \in BExp(V, B)$ by a valuation v (we write $v \models \beta$) is defined inductively as follows: $v \models true$, $v \models e_1 \sim e_2$ iff $v(e_1) \sim v(e_2)$, $v \models empty(b)$ iff $v(b) = \varepsilon$, $v \models g_1 \ and \ g_2$ iff $v \models g_1$ and $v \models g_2$, $v \models g_1 \ or \ g_2$ iff $v \models g_1$ or $v \models g_2$, and $v \models not \ g$ iff $v \not\models g$.

Let $\alpha \in Act(V, B)$ be an action. By $v(\alpha)$ we denote the valuation after execution of the action α on the valuation v. We define $v(\alpha)$ to be the valuation v' defined inductively as follows (by $u.w$ we denote the concatenation of the value u and the sequence of values w, similarly for $w.u$):

- if $\alpha = \varepsilon$, then $v' = v$,
- if $\alpha = (y := exp)$, then $v'(y) = v(exp)$ and $v'(z) = v(z)$ for $z \in (V \cup B) \setminus \{y\}$,
- if $\alpha = get(b, y)$ and $v(b) = u.w$, then $v'(b) = w$, $v'(y) = u$, and $v'(z) = v(z)$ for $z \in (V \cup B) \setminus \{b, y\}$ (the first element of b is removed and its value is assigned to y),
- if $\alpha = put(b, exp)$, $v(b) = w$, and $v(exp) = u$, then $v'(b) = w.u$ and $v'(z) = v(z)$ for $z \in (V \cup B) \setminus \{b\}$ (the value of the expression exp is appended to the end of b),
- if $\alpha = \alpha_1; \alpha_2$, then $v' = v''(\alpha_2)$, where $v'' = v(\alpha_1)$.

There is the runtime error in case of putting en element to a full buffer as well as getting an element out of an empty buffer.

Clocks. Let X be a finite set of real variables, called *clocks*. The set $\Psi(X)$ of all the *clock constraints* over X is defined by the following grammar (we deal with diagonal-free automata):

$$constr ::= true \mid x \sim c \mid constr \ and \ constr$$

where $x \in X$, $c \in \mathbb{N}$, and $\sim \in \{\leq, <, =, >, \geq\}$.

[3] By / we denote integer division and $\mathbb{Z}$ denotes the set of integer numbers, $\mathbb{N}$ — the set of natural numbers, and $\mathbb{R}_+$ — the set of non-negative real numbers.

Clock Valuation. A *clock valuation* is a total mapping $\tau : X \to \mathbb{R}_+$. Satisfiability of a clock constraint $\psi \in \Psi(X)$ by a given clock valuation τ (denoted as $\tau \models \psi$) is defined inductively as follows: $\tau \models true$, $\tau \models x \sim c$ iff $\tau(x) \sim c$, $\tau \models \psi_1$ *and* ψ_2 iff $\tau \models \psi_1$ and $\tau \models \psi_2$. For $\delta \in \mathbb{R}_+$, $\tau + \delta$ denotes the clock valuation τ' such that $\tau'(x) = \tau(x) + \delta$ for all $x \in X$. Moreover, by $\tau[Y := 0]$ we denote the clock valuation τ' such that $\tau'(x) = 0$ for $x \in Y$ and $\tau'(x) = \tau(x)$ for $x \in X \setminus Y$. Finally, by τ^0 we denote the *initial* clock valuation such that $\tau^0(x) = 0$ for all $x \in X$.

3 Intermediate Language

In Intermediate Language (IL for short) a system is described as a set of processes running parallel and communicating with each other via variables (shared memory) or buffers (message passing mechanism). A process is described by terms of states and transitions. There is no explicit time variables in the language, but the time of transition execution can be restricted.

Definition 1. (IL Syntax). *A* program *is a tuple* $\mathcal{P} = (V_P, B, \{P_i \mid 1 \leq i \leq n\})$, *where* V_P *is a set of variables,* B *is a set of buffers,* $n \in \mathbb{N}$, *and a process* P_i *is a tuple* $(id_i, Q_i, q_i^0, \Gamma_i, T_i)$, *where* id_i *is the process name,* Q_i *is a set of control states,* $q_i^0 \in Q_i$ *is the initial control state,* Γ_i *is a set of labels, and* T_i *is a set of transitions of the form* (q, g, d, u, l, a, q'), *where:*

- $q \in Q_i$ *is called the* source*,*
- $q' \in Q_i$ *is called the* target*,*
- $g \in BExp(V_P, B)$ *is called the* guard*,*
- $d \in \{(d_1, d_2), (d_1, d_2], (d_1, \infty), [d_1, d_2), [d_1, d_2], [d_1, \infty) \mid d_1, d_2 \in \mathbb{N}\}$ *is called the* allowed delay*,*
- $u \in \{true, false\}$ *is called the* urgency attribute*,*
- $l \in \Gamma_i$ *is called the* label*, and*
- $a \in Act(V_P, B)$ *is called the* action*.*

Let $t = (q, g, d, u, l, a, q')$. If the allowed delay d is of the form $(d_1, d_2]$, then the transition t must be executed strictly after d_1 and before d_2 time units since the process has reached the location q (similarly for other forms of allowed delays). The urgency attribute u says whether the transition has the priority over time progress. When the urgency attribute is equal to *true* the transition t has to be executed as soon as it becomes enabled. We work under the assumption that for an urgent transition t the delay allowed should be equal to $[0, \infty)$. Notice, that the label l can be either *local* or *synchronous*. A local label is unique in a program and a synchronous one is shared among some processes. It is assumed that buffers and shared variables (used or defined by more than one process) are not defined by actions of synchronous transitions.

Let $T = \bigcup_{i=1}^{n} T_i$, $Q = \bigcup_{i=1}^{n} Q_i$ and $\Gamma = \bigcup_{i=1}^{n} \Gamma_i$. We use $out(q)$ to denote the set of transitions $\{t \in T \mid source(t) = q\}$. By *bufs(a)* $\subseteq B$ we denote the set of buffers that occur in the action a.

Example. The program presented in Fig. 1 is composed of two processes: *producer* and *consumer* communicating via the buffer *buf*. The variables p and c represent portions of data. The transition labels are omitted. The *producer* produces a portion of data (it randomly assigns 0 or 1 to p) which takes from 1 to 2 units of time and passes to the state *transmit*. Next, if the buffer is not full, then *producer* places the portion to the buffer and comes back to the state *produce*. The *consumer* waits until the buffer *buf* is not empty, takes a portion of data from the buffer, and consumes it. The transition from the state *consume* to the state *request* can last an arbitrary amount of time. Note, that a transition marked as urgent must be executed as soon as it becomes enabled.

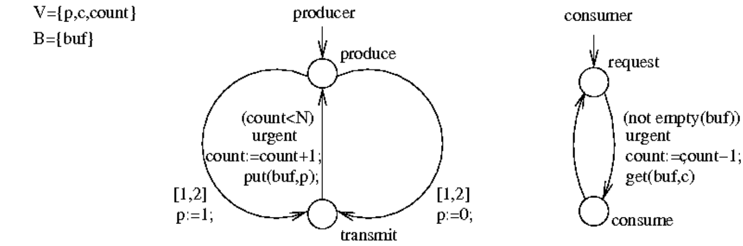

Fig. 1. Producer–Consumer example in Intermediate Language.

Delay valuation. We define a *delay valuation* to be a total mapping $\mu : Q \to \mathbb{R}_+$ which associates a non-negative real number with each location. For $\delta \in \mathbb{R}_+$, $\mu + \delta$ denotes the delay valuation μ', such that $\mu'(q) = \mu(q) + \delta$ for all $q \in Q$. By μ^0 we denote the *initial* delay valuation such that $\mu^0(q) = 0$ for all $q \in Q$. For $Y \subseteq Q$, $\mu[Y := 0]$ denotes the delay valuation μ', such that $\mu'(q) = 0$ for $q \in Y$ and $\mu'(q) = \mu(q)$ for $q \in Q \setminus Y$.

Semantics. A state of a program is a tuple of the form $(q_1, \ldots, q_n, v, \mu)$, where $q_i \in Q_i$ for $1 \leq i \leq n$, $v \in \Omega^{V_P \cup B}$, and $\mu \in {\mathbb{R}_+}^Q$. For a transition $t_i \in T_i$ and a state $s = (q_1, \ldots, q_n, v, \mu)$ we define: $enabled(t_i, s) \triangleq (source(t_i) = q_i) \wedge v \models guard(t_i)$, and $fireable(t_i, s) \triangleq enabled(t_i, s) \wedge \mu(source(t_i)) \in delay(t_i)$.

Processes execute transitions with shared labels synchronously. Local transitions are interleaved. It is required that the transitions with a shared label have to be executed by all the processes containing this label (*multi-synchronization*). By $\Gamma(l) = \{\ 1 \leq i \leq n \mid l \in \Gamma_i\}$ we denote the set of numbers of processes that contain the label l and $(a_j)_{j \in Z}$ denotes 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 indeces.

Definition 2. (IL Semantics). *The semantics of a program* $\mathcal{P} = (V_P, B, \{P_i \mid 1 \leq i \leq n\})$ *with* $P_i = (id_i, Q_i, q_i^0, \Gamma_i, T_i)$ *for an initial valuation* $v^0 : V_P \cup B \to \Omega$ *is a labeled transition system* $\mathcal{S}_P = (S_P, s_P^0, \Lambda_P, \longrightarrow_P)$ *such that:*

- $S_P = Q_1 \times \ldots \times Q_n \times \Omega^{V_P \cup B} \times {\mathbb{R}_+}^Q$ *is the set of* states,
- $s_P^0 = (q_1^0, \ldots, q_n^0, v^0, \mu^0) \in S_P$ *is the initial state,*
- $\Lambda_P = \bigcup_{i=1}^n \Gamma_i \cup \mathbb{R}_+$
- $\longrightarrow_P \subseteq S_P \times \Lambda_P \times S_P$ *is the smallest transition relation defined as follows:*

- *let* $s = (q_1, \ldots, q_n, v, \mu)$ *and* $s' = (q'_1, \ldots, q'_n, v', \mu')$, $s \xrightarrow{l}_P s'$ *iff there exists a set* $T' = \{t_i \mid i \in \Gamma(l)\} \subseteq T$ *such that* $fireable(t_i, s)$, $l = label(t_i)$, $q'_i = target(t_i)$, *for all* $t_i \in T'$, $v' = \alpha(v, (action(t_i))_{i \in \Gamma(l)})$, $\mu' = \mu[\{q'_i\}_{i \in \Gamma(l)} := 0]$ *and* $q'_j = q_j$ *for all* $j \in \{1, \ldots, n\} \setminus \Gamma(l)$,
- *let* $s = (q_1, \ldots, q_n, v, \mu)$ *and* $s' = (q_1, \ldots, q_n, v, \mu + \delta)$, *for* $\delta \in \mathbb{R}_+$ $s \xrightarrow{\delta}_P s'$ *iff* $\delta > 0$ *and* $\neg enabled(t, s) \vee (\neg urgent(t, s) \wedge \neg(fireable(t, s) \wedge \neg fireable(t, s')))$ *holds for all* $t \in T$ *or* $\delta = 0$.

Initially, all the buffers are empty and all the variables have some initial values. At a state $s = (q_1, \ldots, q_n, v, \mu)$ the system can: either execute a local fireable transition or a set of synchronous fireable transitions (the delay values of location of processes performing transitions are reset to zero), or let time δ pass and move to the state $(q_1, \ldots, q_n, v, \mu + \delta)$, unless there is an enabled transition with the urgency attribute set or the time passage would make any fireable transition not fireable.

For $(q_1, \ldots, q_n, v, \mu) \in S_P$, let $(q_1, \ldots, q_n, v, \mu) + \delta$ denote $(q_1, \ldots, q_n, v, \mu + \delta)$. An $s{-}run$ of $\mathcal{P}$ is an finite sequence: $s_0 \xrightarrow{\delta_0}_P s_0 + \delta_0 \xrightarrow{l_0}_P s_1 \xrightarrow{\delta_1}_P s_1 + \delta_1 \xrightarrow{l_1}_P \ldots$, where $s_0 = s \in S$, $l_i \in \Gamma$, and $\delta_i \in \mathbb{R}_+$ for each $i \geq 0$. For simplicity in our work we consider only infinite sequences since the finite ones correspond to erroneous situations.

4 Timed Automata

Timed automata with discrete data (TADD for short) are timed automata [2] enriched with additional integer variables.

Definition 3. (TADD Syntax). *A* timed automaton with discrete data *is a tuple* $\mathcal{TA} = (\Sigma, L, \varrho^0, V_A, X, E, \mathcal{I})$, *where*

- Σ *is a finite set of* labels,
- L *is a finite set of* locations,
- ϱ^0 *is the* initial location,
- V_A *is a finite set of integer variables,*
- X *is a finite set of clocks,*
- $E \subseteq L \times \Sigma \times BExp(V_A) \times \Psi(X) \times \{true, false\} \times Act(V_A) \times 2^X \times L$ *is a* transition relation, *and*
- $\mathcal{I} : L \longrightarrow \Psi(X)$ *is an* invariant *function.*

An element $e = (\varrho, \lambda, \phi, \psi, \upsilon, \alpha, Y, \varrho')$ of E denotes a transition from the location ϱ to the location ϱ', where λ is the label of the transition e, ϕ and ψ define the enabling conditions (the guard and the clock constraint) for e, α is the action to be executed, Y is the set of clocks to reset, and υ is the urgency attribute – when it is set, the transition t has to be executed as soon as it is enabled (e is enabled if the automaton is in the location ϱ and the guard ϕ evaluates to $true$). We assume that clock constraints of urgent transitions are equal to $true$. The invariant function assigns to each location $\varrho \in L$ a clock constraint defining the condition under which the automaton can stay in ϱ. It is assumed that there are only upper time bounds in invariants.

Definition 4. (TADD Semantics) Semantics *of a timed automaton with discrete data* $\mathcal{TA} = (\Sigma, L, \varrho^0, V_A, X, E, \mathcal{I})$ *for an initial valuation* $v^0 : V_A \to \mathbb{Z}$ *is a labeled transition system* $\mathcal{S}_A = (S_A, s_A^0, \Lambda_A, \longrightarrow_A)$*, where:*

- $S_A = \{(\varrho, v, \tau) \mid \varrho \in L \wedge v \in \mathbb{Z}^{V_A} \wedge \tau \in \mathbb{R}_+{}^X \wedge \tau \models \mathcal{I}(\varrho)\}$ *is the set of* states*,*
- $s_A^0 = (\varrho^0, v^0, \tau^0) \in S_A$ *is the* initial state*,*
- $\Lambda_A = \Sigma \cup \mathbb{R}_+$ *is the set of labels,*
- $\longrightarrow_A \subseteq S_A \times \Lambda_A \times S_A$ *is the smallest transition relation defined by the following rules:*
  - $(\varrho, v, \tau) \xrightarrow{\lambda}_A (\varrho', v', \tau')$ *iff there exists a transition* $e = (\varrho, \lambda, \phi, \psi, \upsilon, \alpha, Y, \varrho') \in E$ *such that* $v \models \phi$*,* $\tau \models \psi$*,* $v' = v(\alpha)$ *and* $\tau' = \tau[Y := 0] \models \mathcal{I}(\varrho')$*,*
  - $(\varrho, v, \tau) \xrightarrow{\delta}_A (\varrho, v, \tau{+}\delta)$ *iff* $\tau{+}\delta \models \mathcal{I}(\varrho)$ *and for all transitions* $e = (\varrho, \lambda, \phi, \psi, \upsilon, \alpha, Y, \varrho') \in E$*,* $\upsilon = false$ *or* $v \not\models \phi$ *or* $\tau[Y := 0] \not\models \mathcal{I}(\varrho')$*.*

Initially, all variables have theirs initial values and all clocks are set to 0. Being in a state $s = (\varrho, v, \tau)$ the system can: either execute an enabled transition e and move to the state $s' = (\varrho', v', \tau')$, where ϱ' is the target location of e (the variables valuation is changed according to the action of e and the clocks included in the reset set of e are set to 0), or let time δ pass and move to the state $(\varrho, v, \tau + \delta)$, as long as $\tau + \delta$ satisfy $\mathcal{I}(\varrho)$ and no urgent transition can be executed.

For $(\varrho, v, \tau) \in S_A$, let $(\varrho, v, \tau) + \delta$ denote $(\varrho, v, \tau + \delta)$. An $s{-}run$ π of $\mathcal{TA}$ is an infinite sequence: $s_0 \xrightarrow{\delta_0}_A s_0 + \delta_0 \xrightarrow{\lambda_0}_A s_1 \xrightarrow{\delta_1}_A s_1 + \delta_1 \xrightarrow{\lambda_1}_A \ldots$, where $s_0 = s \in S$, $\lambda_i \in \Sigma$ and $\delta_i \in \mathbb{R}_+$ for each $i \geq 0$.

Parallel Composition. Let $\mathcal{TA}_1, \ldots \mathcal{TA}_n$ be a set of TADD and $\Sigma(\lambda) = \{1 \leq i \leq n \mid \lambda \in \Sigma_i\}$ denote a set of numbers of automata containing the label λ.

In the parallel compostition of automata, transitions with a shared label are executed synchronously by all automata containing this label. To obtain clear semantics of variables updating it is necessary to fix the order of actions in the 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 *input* transitions. We assume additionally that input transitions do not update shared variables. By ϵ we denote an empty string which concatenated with a character λ gives λ. Now a transition label is of the form: $\lambda!$, $\lambda?$ or λ, where $\lambda \in \bigcup_{i=1}^n \Sigma_i$.

Definition 5. (Parallel Composition of TADD). *Let* $\mathcal{TA}_1, \ldots \mathcal{TA}_n$ *be a set of TADD, where* $\mathcal{TA}_i = (\Sigma_i \times \{?, !, \varepsilon\}, Q_i, q_i^0, V_i, X_i, E_i, \mathcal{I}_i)$*. A parallel composition* $\mathcal{TA}_1 \parallel \mathcal{TA}_2 \parallel \ldots \parallel \mathcal{TA}_n$ *is the TADD* $\mathcal{TA} = (\Sigma, L, \varrho^0, V_A, X, E, \mathcal{I})$*, where:* $\Sigma = \bigcup_{i=1}^n \Sigma_i$*,* $L = \prod_{i=1}^n L_i$*,* $\varrho^0 = (\varrho_1^0, \ldots, \varrho_n^0)$*,* $V_A = \bigcup_{i=1}^n V_i$*,* $X = \bigcup_{i=1}^n X_i$*,* $\mathcal{I}(\varrho_1, \ldots, \varrho_n) = \bigwedge_{i=1}^m \mathcal{I}_i(\varrho_i)$*, and relation* E *is defined as follows:* $((\varrho_1, \ldots, \varrho_n), \lambda, \phi, \psi, \upsilon, \alpha, Y, (\varrho_1', \ldots, \varrho_n')) \in E$ *iff*

- *there exists* $j \in \Sigma(\lambda)$ *such that* $\lambda_j = \lambda!$ *and for all* $i \in \Sigma(\lambda)$*,* $(\varrho_i, \lambda, \phi_i, \psi_i, \upsilon_i, \alpha_i, Y_i, \varrho_i') \in E_i$*,* $\phi = \bigwedge_{i \in \Sigma(\lambda)} \phi_i$*,* $\psi = \bigwedge_{i \in \Sigma(\lambda)} \psi_i$*,* $\upsilon = \bigvee_{i \in \Sigma(\lambda)} \upsilon_i$*,* $\alpha = \alpha_j(\alpha_i)_{i \in \Sigma(\lambda) \setminus \{j\}}$*,* $Y = \bigcup_{i \in \Sigma(\lambda)} Y_i$ *and for all* $i \in \{1, \ldots, n\} \setminus \Sigma(\lambda)$*,* $\varrho_i' = \varrho_i$*,*
- *for all* $i \in \Sigma(\lambda)$ *and* $(\varrho_i, \lambda, \phi_i, \psi_i, \upsilon_i, \alpha_i, Y_i, \varrho_i') \in E_i$*, where* $\phi = \bigwedge_{i \in \Sigma(\lambda)} \phi_i$*,* $\psi = \bigwedge_{i \in \Sigma(\lambda)} \psi_i$*,* $\upsilon = \bigvee_{i \in \Sigma(\lambda)} \upsilon_i$*,* $\alpha = (\alpha_i)_{i \in \Sigma(\lambda)}$*,* $Y = \bigcup_{i \in \Sigma(\lambda)} Y_i$ *and for all* $j \in \{1, \ldots, n\} \setminus \Sigma(\lambda)$*,* $\varrho_j' = \varrho_j$*.*

5 Translation from Intermediate Language to TADD

Since the formalisms are similar, the translation is quite natural. Only two aspects, namely time restrictions and communication via buffers, need more attention. The main idea of the translation is the following: for each process in the Intermediate Language program a TADD is constructed. Each control state of the process has exactly one corresponding location in the automaton. Two additional locations represent the case of putting an element to a full buffer ($overflow$) and getting an element from empty buffer ($empty$). If there is a transition from a control state q_1 to a control state q_2 in the process, then there is at least one transition between corresponding locations in the automaton.

Each timed automaton for a process has one clock. The clock is reset with each transition. The time constraint on the clock of a transition in the automaton is derived from allowed delays of its corresponding transition in the process.

Buffers. A buffer $b \in B$ is modeled by $size(b) + 3$ variables. Let V_b be the set $\{cnt_b, out_b, in_b, b1, \ldots, bsize(b)\}$. The variable cnt_b represents the number of elements stored in the buffer at the moment, out_b indicates its first element and in_b points the place, where a new element will be stored. Variables $b1, \ldots, bsize(b)$ model the buffer content (see Fig. 2).

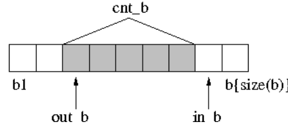

Fig. 2. Implementation of a buffer.

If an action of a process transition contains one operation $put(b, exp)$ or $get(b, y)$, then in the automaton there are $size(b)$ corresponding transitions of the form depicted in Fig. 3. For sake of clarity we present the definition of the translation for a program, where there is at most one operation get or put on a given buffer $b \in B$ in the action of each transition.

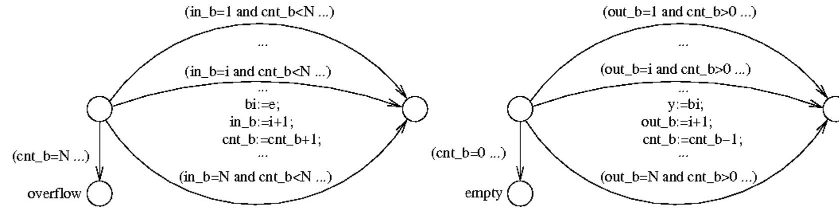

Fig. 3. Counterparts of operations $put(b, e)$ and $get(b, y)$ for $size(b) = N$.

Time constraints. We present the translation under the assumption that all transitions that are not urgent and go out of one location have the same upper bounds of the allowed delays. Let x_i be the clock of i-th process. We define two family of functions: $constr_i : T_i \rightarrow \Psi(\{x_i\})$ and $upper_constr_i : Q_i \rightarrow \Psi(\{x_i\})$ for $1 \leq i \leq n$, which for each transition (and

each control state, resp.) construct the suitable clock constraints:

$$constr_i(t) = \begin{cases} d_1 < x_i \wedge x_i < d_2 & \text{if } delay(t) = (d_1, d_2), \\ d_1 < x_i \wedge x_i \leq d_2 & \text{if } delay(t) = (d_1, d_2], \\ d_1 < x_i & \text{if } delay(t) = (d_1, \infty), \\ d_1 \leq x_i \wedge x_i < d_2 & \text{if } delay(t) = [d_1, d_2), \\ d_1 \leq x_i \wedge x_i \leq d_2 & \text{if } delay(t) = [d_1, d_2], \\ d_1 \leq x_i & \text{if } delay(t) = [d_1, \infty), \text{ where } d_1 \neq 0, \\ true & \text{if } delay(t) = [0, \infty). \end{cases}$$

$$upper_constr_i(q) = \begin{cases} x_i < d_2 & \text{if } \exists_{t \in out(q)} \exists_{d_1 \in \mathbb{N}}\ delay(t) = (d_1, d_2) \vee delay(t) = [d_1, d_2) \\ x_i \leq d_2 & \text{if } \exists_{t \in out(q)} \exists_{d_1 \in \mathbb{N}}\ delay(t) = (d_1, d_2] \vee delay(t) = [d_1, d_2] \\ true & \text{if } \forall_{t \in out(q)} \exists_{d_1 \in \mathbb{N}}\ delay(t) = (d_1, \infty) \vee delay(t) = [d_1, \infty) \end{cases}$$

Definition 6. (The set of TADD for Intermediate Language program). *For a program $\mathcal{P} = (V_P, B, \{P_i \mid 1 \leq i \leq n\})$, where $P_i = (id_i, Q_i, q_i^0, \Gamma_i, T_i)$, we define a set $\mathcal{TA}_1, \ldots \mathcal{TA}_n$ of TADD, where $\mathcal{TA}_i = (\Sigma_i, L_i, \varrho_i^0, V_A, X_i, E_i, \mathcal{I}_i)$ as follows:*

- $\Sigma_i = \Gamma_i$,
- $L_i = Q_i \cup \{overflow_i, empty_i\}$,
- $\varrho_i^0 = q_i^0$,
- $V_A = V_P \cup \bigcup_{b \in B} V_b$[4],
- $X = \{x_i\}$, *if there exists a transition in the process with a delay different than* $[0, \infty)$, *otherwise* $X = \emptyset$,
- *the relation E_i is defined as follows: for each transition $t = (q, g, d, u, l, a, q') \in T_i$*
  - *if* bufs(a)= $\emptyset$, *then there is one transition $e = (\varrho, \lambda, \phi, \psi, \upsilon, \alpha, Y, \varrho') \in E_i$, where $\varrho = q$, $\lambda = l$, $\phi = g'$, where g' is the guard g with each expression of the form $empty(b)$ replaced with $cnt_b = 0$, $\psi = constr(t)$, $\upsilon = u$, $\alpha = a$, $Y = \{x_i\}$, $\varrho' = q'$,*
  - *if* bufs(a)= $\{b_1, \ldots, b_m\} \neq \emptyset$, *then there is $size(b_1) * \ldots * size(b_m) + m$ transitions $e_{jk_j} = (\varrho, \lambda, \phi_{jk_j}, \psi, \upsilon, \alpha_{jk_j}, \varrho'_{jk_j}) \in E_i$ for $1 \leq j \leq m$ and $0 \leq k_j \leq size(b_j)$, where*
    - $\varrho = q$, $\lambda = l$, $\psi = constr(t)$, $\upsilon = u$, $Y = \{x_i\}$,
    - $\phi_{jk_j} = (g'$ *and* $\phi'_{jk_j})$, *where g' is the guard g, where each expression of the form $empty(b)$ is replaced with $cnt_b = 0$,*
    - *if the operation of the form $get(b_j, y)$ for some $y \in V_P$ occurs in a, then $\phi'_{j0} = (cnt_b_j = 0)$, $\varrho'_{j0} = empty_i$, $\phi'_{jk_j} = (cnt_b_j > 0$ and $out_b_j = k_j)$, and $\varrho'_{jk_j} = q'$ for $1 \leq k_j \leq size(b_j)$, otherwise ($put(b_j, e)$ for some $e \in Exp(V_P)$ occurs in a) $\phi'_{j0} = (cnt_b_j = size(b_j))$, $\varrho'_{j0} = overflow_i$, $\phi'_{jk_j} = (cnt_b_j < size(b_j)$ and $in_b_j = k_j)$, and $\varrho'_{jk_j} = q'$ for $1 \leq k_j \leq size(b_j)$;*
    - $\alpha_{jk_j} = a'$, *where a' is a with each operation $put(b_j, e)$ for $e \in Exp(V_P)$ replaced with the sequence of operations: $b_j k_j := e$; $in_b_j := k_j + 1$; $cnt_b_j := cnt_b_j + 1$; and operation $get(b_j, y)$ replaced with the sequence of operations: $y := b_j k_j$; $out_b_j := k_j + 1$; $cnt_b_j := cnt_b_j - 1$;,*
- *for $\varrho \in L_i$ the invariant function $\mathcal{I}(\varrho) = upper_constr(q)$, where q is the control state corresponding to ϱ.*

---

[4] Some variables are not used by the automaton.

Example. For the Intermediate Language program from Sect. 3, where $size(buf) = N = 2$, the set of two automata is constructed as presented in Fig. 4.

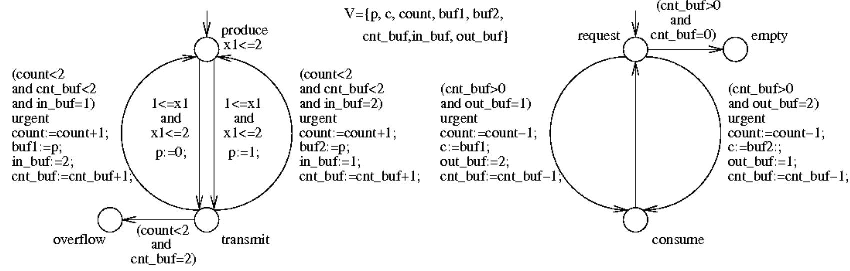

Fig. 4. Timed automata for the Producer–Consumer example.

Verification. Let $\mathcal{S}_P = (S_P, s_P^0, \Lambda_P, \longrightarrow_P)$ be the labeled transition system of a program $\mathcal{P} = (V_P, B, \{P_i \mid 1 \leq i \leq n\})$, where $P_i = (id_i, Q_i, q_i^0, \Gamma_i, T_i)$. For $\mathcal{P}$ we can define *propositional variables* of the form $p_{i.q}$, where $1 \leq i \leq n$ and $q \in Q_i$, $p_{empty(b)}$ for $b \in B$, or $p_{e_1 \sim e_2}$, where $e_1, e_2 \in Exp(V_P)$ and $\sim$ is a relational operator. We denote the set of all propositional variables by PV.

In order to reason about systems represented as Intermediate Language programs we define a *labeling* functions $\mathcal{V}_P : S_P \rightarrow 2^{PV}$. For $s = (q_1, \ldots, q_n, v, \mu) \in S_P$: $p_{e_1 \sim e_2} \in \mathcal{V}_P(s)$ iff $v \models e_1 \sim e_2$, $p_{empty(b)} \in \mathcal{V}_P(s)$ iff $v(b) = \varepsilon$, and $p_{i.q} \in \mathcal{V}_P(s)$ iff $q_i = q$. The *model* of a program is the pair $\mathcal{M}_P = (\mathcal{S}_P, \mathcal{V}_P)$.

Let $\{\mathcal{TA}_1, \ldots, \mathcal{TA}_n\}$, where $\mathcal{TA}_i = (\Sigma_i, L_i, \varrho_i^0, V_A, X_i, E_i, \mathcal{I}_i)$, be a set of TADD constructed for the program $\mathcal{P}$ according to Def. 6 and $\mathcal{S}_A = (S_A, s_A^0, \Lambda_A, \longrightarrow_A)$ denote the labeled transition system of a parallel composition of the automata of this set. Notice, that $V_P \subseteq V_A$ and for each control state $q \in Q_i$ there exists the corresponding location $l \in L_i$ for $1 \leq i \leq n$. So, for a given set of propositional variables PV we can define a *labeling* function $\mathcal{V}_A : S_A \rightarrow 2^{PV}$ such that for $s' = (\varrho_i, \ldots, \varrho_n, v', \tau) \in S_A$, $p_{e_1 \sim e_2} \in \mathcal{V}_A(s')$ iff $v' \models e_1 \sim e_2$, $p_{empty(b)} \in \mathcal{V}_A(s')$ iff $v'(cnt_b) = 0$, and $p_{i.q} \in \mathcal{V}_A(s')$ iff $\varrho_i = \varrho$, where ϱ is the location corresponding to the control state q. Let $\mathcal{M}_A = (\mathcal{S}_A, \mathcal{V}_A)$. Our aim is to show that $\mathcal{M}_P$ and $\mathcal{M}_A$ are strongly bisimilar [3].

For each buffer b we define the function $seq_b : V_b \times V_A^{\mathbb{Z}} \rightarrow \mathbb{Z}^*$, which constructs a sequence of values that corresponds to the content of the buffer b. Let $N = size(b)$ and $V_b = \{cnt_b, in_b, out_b, b1, \ldots, bN\}$. If $v(cnt_b) = 0$ then $seq_b(V_b, v) = \varepsilon$, otherwise $seq_b(V_b, v) = u1 \ldots u_k$, where $k = v(cnt_b)$ and $u_j = v(b\{j \mod N{+}1\})$ for $in_b \leq j \leq v(in_b){+}v(cnt_b){-}1$.

Definition 7. *Let $\cong \subseteq S_P \times S_A$ be the relation s.t. for each $s = (q_1, \ldots, q_n, v, \mu) \in S_P$ and each $s' = (\varrho_1, \ldots, \varrho_n, v', \tau') \in S_A$ we have $s \cong s'$ iff the following holds:*

1. *$q_i = \varrho_i$ for each $1 \leq i \leq n$,*
2. *$v(y) = v'(y)$ for each $y \in V_P \subseteq V_A$ and $v(b) = seq(V_b, v')$ for each $b \in B$,*
3. *$\mu(q_i) = \tau(x_i)$ for each $1 \leq i \leq n$.*

Lemma 1. *The relation $\cong \subseteq S_P \times S_A$ is a strong bisimulation between $\mathcal{M}_P$ and $\mathcal{M}_A$.*

Proof. (a sketch) Let $s = (q_1, \ldots, q_n, v, \mu) \in S_P$ and $s' = (\varrho_1, \ldots, \varrho_n, v', \tau') \in S_A$. It is easy to check that $s_P^0 \cong {s_A^0}'$. From Def.7.1-2 it follows that $\mathcal{V}_P(s) = \mathcal{V}_A(s')$ for $s \cong s'$. ($\Rightarrow$) Let $s_1 \cong s_1'$ and $s_1 \xrightarrow{a}_P s_2$. We will show that there exists a state s_2' such that $s_1' \xrightarrow{a}_A s_2'$ and $s_2 \cong s_2'$. Let us consider two cases. Case 1: $a \in \Gamma$. Let T' be a set of executed transitions (with label a). For each $t \in T' \cap T_i$ there exists $e \in E_i$ corresponding to t. From Def. 7 it follows that e can be executed (we leave the details to the reader). Clearly, after the execution of corresponding transitions Item 1 of Def. 7 is satisfied for s_2 and s_2'. Next, by Def. 6 the action of e is "equivalent" to the action of t, which means that values of variables and contents of buffers are changed in the same way. It follows that after the execution of the "equivalent" actions Item 2 is satisfied for s_2 and s_2'. During an action transition, delays of control states which are target of transitions from the set T' and values of clocks of automata performing the corresponding transitions are set to 0. The rest of delay and clocks values stays unchanged. Thus, Item 3 is also satisfied for s_2 and s_2'.

Case 2: $a \in \mathbb{R}_+$, which means that the transition represents the passage of time. It is easy to see from the definition of time constraints that a time units can also pass for the set of TADD. Clearly, during a timed transition, control states, locations and valuations are not changed and all the delay values and all the clocks values are incremented by a. Hence, all the conditions of Def. 7 are satisfied for s_2 and s_2'.

The proof of ($\Leftarrow$) can be done in the similar way.

It follows from [3] that if there exists a strong bisimulation between two models, then these models satisfy the same CTL$_{\text{-X}}$* [4] formulas. Thus, a consequence of Lemma 1 is that the model of a program satisfies an CTL$_{\text{-X}}$* formula if and only if the model of the set of TADD constructed for the program satisfies the formula.

6 Experimental results

We present experimental results obtained with the verification tool VerICS on the machine equipped with the processor Intel Pentium 3 GHz, 2 GB of main memory, and the Linux Ubuntu 7.04 operating system. We compare amounts of time and memory consumed by the translation of Intermediate Language program to a network of classical timed automata [6] (TA) and to timed automata with discrete data (TADD), and then to a propositional formula by the bounded model checking modules of VerICS for timed automata [10] and for TADD [9] and satisfiability verification of the formula by SAT solver MiniSat.

Let us first consider the example presented in Sect. 3 (PC). We have checked whether *Consumer* can be in its control state *consume* while *count* $= N$ for various values of N (sizes of the buffer).

To validate the technique we have also performed experiments with a case study known from the literature. For Fischer Mutual Exclusion Protocol (FMEP) [1] shown in Fig. 5 the following formulae have been tested (N is the number of processes).

$$\varphi = EF(\bigvee_{1 \le i < j \le N} (p_{i.critical_i} \wedge p_{j.critical_j}))$$
$$\psi = EF(\bigvee_{1 \le i_1 < \ldots < i_{N-1} \le N} (p_{i.critical_i} \wedge \ldots \wedge p_{i_{N-1}.critical_{i_{N-1}}}))$$

In all the cases presented, properties being verified are violated, that is erroneous scenarios are found and the length of the shortest counterexample (depth) is shown.

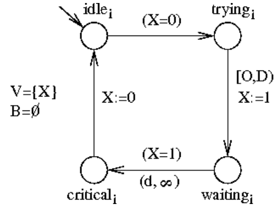

Fig. 5. ith process of Fischer Mutual Exclusion Protocol.

Selected results of the experiments are presented in Fig. 6.

				translation		bmc		minisat	
example	parameters	version	depth	sec	MB	sec	MB	sec	MB
PC	$N = 1$	*TA*	10	0.42	4.12	0.08	2.90	0.02	3.16
		TADD	10	0.03	0.50	0.13	2.40	0.02	2.98
	$N = 2$	*TA*	14	0.84	4.12	0.14	4.32	0.11	4.47
		TADD	14	0.03	0.50	0.24	3.20	0.06	3.93
	$N = 4$	*TA*	22	0.94	4.12	0.49	8.94	0.57	8.81
		TADD	22	0.03	0.50	0.54	4.80	0.27	6.13
	$N = 6$	*TA*	30	6.87	4.12	1.89	18.21	2.90	17.22
		TADD	30	0.03	0.50	0.97	7.20	0.79	9.43
FMEP, φ	$N = 2$	*TA*	12	0.21	4.15	0.06	2.77	0.02	2.98
		TADD	12	0.03	0.50	0.04	1.80	0.01	2.12
	$N = 4$	*TA*	12	1.15	4.15	0.12	3.93	0.18	4.04
		TADD	12	0.03	0.50	0.10	2.10	0.07	2.55
	$N = 6$	*TA*	12	73.45	4.15	0.20	5.34	0.49	5.39
		TADD	12	0.03	0.50	0.16	2.50	0.03	3.00
	$N = 8$	*TA*	-	-	-	-	-	-	-
		TADD	12	0.04	0.50	0.24	2.90	0.24	3.51
FMEP, ψ	$N = 4$	*TA*	26	1.15	4.15	0.50	11.79	5.73	13.16
		TADD	26	0.03	0.50	0.21	2.90	1.13	3.90
	$N = 6$	*TA*	54	73.45	4.15	4.53	79.07	4431.45	952.84
		TADD	54	0.03	0.50	0.80	6.10	1091.69	77.84
	$N = 8$	*TA*	-	-	-	-	-	-	-
		TADD	-	0.04	0.50	-	-	-	-

Fig. 6. Selected experimental results.

The experiments prove that the verification of the IL code is more effective using the method descibed in this paper in comparison with verification of the code translated into classical timed automata.

One observation is evident. The new translation itself is instant and consumes minimal computer resources. On the contrary, the translation to classical time automata was a slow, complicated process. We can see that the old translation of the FMEP program for 8 processes did not succeed,

while the new translation was very fast. This follows from the fact that the structure of IL programs and networks of TADD automata is very similar.

The other observation is that in most cases both the processes of creation and verifying the appropriate formula take less time and memory when using the new method. Moreover, the larger is the model, the differences are more clearly visible. Thus, the translation described in this paper proves very useful while verifying large systems.

7 Conclusions

In the paper we have presented how to translate Intermediate Language programs to the sets of timed automata with discrete data. We have shown that the method of verification of Intermediate Language programs via translation to this formalism is more efficient than verification via translation to classical timed automata.

References

1. R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proc. of the 13th IEEE Real-Time Systems Symposium (RTSS'92)*, pages 157–166. IEEE Computer Society, 1992.
2. R Alur and D Dill. Automata for modelling real-time systems. In *Proc. of the Int. Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *LNCS*, pages 322–335. Springer-Verlag, 1990.
3. M. C. Browne, E. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1/2):115–131, 1988.
4. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons for branching-time temporal logic. In *Proc. of Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.
5. P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Półrola, M. Szreter, B. Woźna, and A. Zbrzezny. VerICS: A tool for verifying Timed Automata and Estelle specifications. In *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 278–283. Springer-Verlag, 2003.
6. A. Doroś, A. Janowska, and P. Janowski. From specification languages to Timed Automata. In *Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'02)*, volume 161(1) of *Informatik-Berichte*, pages 117–128. Humboldt University, 2002.
7. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.
8. P. Pettersson and K. G. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.
9. A. Półrola and A. Zbrzezny. Sat-based reachability checking for timed automata with discrete data. In *Proc. of the Int. Workshop on Concurency, Specification and Programming (CSP'06)*, volume 206(2) of *Informatik Berichte*, pages 207–218. Humboldt University, 2006.
10. B. Woźna, A. Zbrzezny, and W. Penczek. Checking reachability properties for timed automata via SAT. *Fundamenta Informaticae*, 55(2):223–241, 2003.