Journal of Software Engineering and Applications, 2016, 9, 452-478
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Specification and Verification of Dynamically Reconfigurable Systems Using Dynamic Linear Hybrid Automata

Ryo Yanase[1], Tatsunori Sakai[1], Makoto Sakai[1], Satoshi Yamane[2]

[1]Graduate School of Natural Science & Technology, Kanazawa University, Kanazawa, Japan
[2]Institute of Science and Engineering, Kanazawa University, Kanazawa, Japan
Email: ryanase@csl.ec.t.kanazawa-u.ac.jp, tsakai@csl.ec.t.kanazawa-u.ac.jp, sakai@csl.ec.t.kanazawa-u.ac.jp, syamane@is.t.kanazawa-u.ac.jp

How to cite this paper: Yanase, R., Sakai, T., Sakai, M. and Yamane, S. (2016) Specification and Verification of Dynamically Reconfigurable Systems Using Dynamic Linear Hybrid Automata. *Journal of Software Engineering and Applications*, **9**, 452-478.
http://dx.doi.org/10.4236/jsea.2016.99030

Received: July 15, 2016
Accepted: September 27, 2016
Published: September 30, 2016

Abstract

A dynamically reconfigurable system can change its configuration during operation, and studies of such systems are being carried out in many fields. In particular, medical technology and aerospace engineering must ensure system safety because any defect will have serious consequences. Model checking is a method for verifying system safety. In this paper, we propose the Dynamic Linear Hybrid Automaton (DLHA) specification language and show a method to analyze reachability for a system consisting of several DLHAs.

Keywords

Formal Method, Model Checking, Hybrid Automata, Embedded Systems, Dynamically Reconfigurable Systems

1. Introduction

Dynamically reconfigurable systems can change their configuration during operation. Such systems are being used in a number of areas [1]-[4] of an apparatus that involves human lives or expensive manufactured goods (e.g., in medical or aerospace engineering). Here, it is very important to guarantee safety. The major methods of checking system safety include simulation and testing; however, it is difficult for them to ensure safety precisely, since large systems can have infinite state spaces. In such a case, model checking that performs exhaustive searches is a more effective method.

In this paper, we propose the *Dynamic Linear Hybrid Automaton* (DLHA) specification language for describing dynamically reconfigurable systems and provide a reachability analysis algorithm for verifying system safety.

1.1. Our Proposal

The target of our research is an embedded system in which a CPU and dynamically reconfigurable hardware, e.g., DRP (Dynamically Reconfigurable Processor) or dynamically reconfigurable FPGA (Field-Programmable Gate Array) [5] [6] operate cooperatively. The DRP is a coarse-grained programmable processor developed by NEC Corporation [4], and it manages both the power conservation and miniaturization. The DRP is used to accelerate the computations of a general purpose CPU through cooperative operations, and it has the following features:

- Dynamic creation/destruction of functions: when a process occurs, the DRP constitutes a private circuit for processing it. The circuit configuration is released after the process finishes.
- Hybrid property: the operation frequency changes whenever a context switch occurs.
- Parallel execution: the DRP executes several processes on the same board at the same time.
- Queue for communication: the DRP asynchronously receives processing requests from the CPU.

1.1.1. Specification

We devised the following new specification techniques for dynamically reconfigurable systems consisting of CPUs and DRPs:

- We use linear hybrid automata [7] describing changes in the operating frequency.
- We use linear hybrid automata that have creation/destruction events describing dynamic creations and destructions of configuration components.
- We use FIFO queues describing asynchronous communication.

We developed a new specification language (DLHA) based on a linear hybrid automaton with both creation/destruction events and unbounded FIFO queues. DLHA is different from existing research in the following points:

- V. Varshavsky and J. Esparza proposed the GALA (Globally Asynchronous - Locally Arbitrary) modeling approach including timed guards [8]. This approach cannot describe hybrid systems since it is the specification language based on discrete systems. Thus, GALA cannot represent changes in operating frequency.
- S. Minami and others have specified a dynamically reconfigurable system using linear hybrid automata and have verified it by using a model checker, H YT ECH [9]. Since linear hybrid automata cannot describe changes to the configuration and asynchronous communications, the system has been specified as a static system. Therefore, the specification presented in their work is unsuitable for representing dynamically reconfigurable systems. Moreover, they verified only the *schedulability property* of the system, whereas we have verified several other properties in our work.

1.1.2. Verification Method

The originality of our work on the verification method is twofold:

- Our method targets systems that dynamically change their configurations, which is something the existing work, such as H YT ECH, has studied. We extend the syntax and semantics of linear hybrid automata with special actions called *creation actions* and *destruction actions*. We define a state in which an automaton does not exist and transitions for creation and destruction.
- Our method is a comprehensive symbolic verification for hybrid properties, FIFO queues and creation/destruction of tasks.

1.1.3. Experiments on Verifying Dynamically Reconfigurable Systems

For the experiments, we specified a dynamically reconfigurable embedded system consisting of a CPU and DRP, and verified some of its important features. This is the first time that specification and verification of dynamic changes have been tried in a practical case.

1.2. Related Work

Here, we describe related work and how it differs from our work.

- P. C. Attie and N. A. Lynch specified systems whose components are dynamically created/destroyed by using I/O automata [10]. I/O automata cannot describe changes in variables, for example, changes in the clock and operating frequency.
- H. Yamada and others proposed hierarchical linear hybrid automata for specifying dynamically reconfigurable systems [11]. They introduced concepts such as class, object, etc., to the specification language. However, as the scale of the system to be specified increases, the representation and method of analysis in the verification stage tend to be complex.
- B. Boigelot and P. Godefroid specified a communication protocol in terms of finite-state machines and unbounded FIFO buffers (queues), and they verified it [12]. Since the finite-state machine also cannot describe changes in variables, it is unsuitable in our case.
- A. Bouajjani and others proposed a reachability analysis for pushdown automata and a symbolic reachability analysis for FIFO-channel systems [13] [14]. However, since their analysis don't provide for continuous changes in variables, in languages cannot be used for designing hybrid systems.

2. Dynamic Linear Hybrid Automaton

2.1. Preliminaries

Definition 1 (Constraint). *Let V be a finite set of variables. A constraint* ϕ *on V is defined as*

$$\phi ::= true \mid x \sim e \mid x - y \sim e \mid \phi_1 \wedge \phi_2,$$

where $x, y \in V$, $e \in \mathbb{Q}$, ϕ_1 and ϕ_2 are *constraints* on *V*, and $\sim \in \{=, <, >, \leq, \geq\}$. $\Phi(V)$ denotes the set of all constraints on *V*.

Definition 2 (Flow condition). *Let* $V = \{x_1, \cdots, x_n\}$ *be a finite set of* (*real-valued*) *variables. A flow condition f on V is defined as*

$$f ::= \dot{x}_1 = d_1 \wedge \cdots \wedge \dot{x}_n = d_1,$$

where $d_1, \cdots, d_n \in \mathbb{Q}$. $F(V)$ is the set of all flow conditions.

For each variable *x*, we use the dotted variable $\dot{x}$, to denote the first derivative of *x*.

Definition 3 (Update Expression). *Let V be a finite set of variables. An update expression upd on V is defined as*

$$upd ::= x := c \mid x := x + c,$$

where $x \in V$ and $c \in \mathbb{Q}$. $UPD(V)$ is the set of all update expressions can be written.

2.2. Syntax

A dynamic linear hybrid automaton (DLHA) is a tuple $(L, V, Inv, Flow, Act, T, t_0, T_d)$, where

- L is a finite set of locations.
- V is a finite set of (real-valued) variables.
- $Inv : L \to \Phi(V)$ is a function that assigns a constraint to each location.
- $Flow : L \to F(V)$ is a function that assigns a flow condition to each location.
- $Act = Act_{in} \cup Act_{out} \cup Act_{\tau}$ is a finite set of *actions*.

- Act_{in} is a finite set of *input actions*, and each input action has the form $a?$. An input action $m?$ denotes receiving the message *m*.

- Act_{out} is a finite set of *output actions*, and each output action has the form $a!$. An output action $m!$ denotes sending the message *m* to each DLHA.

- Act_{τ} is a finite set of *internal actions* that denote changing a state of a DLHA.

Moreover, we formalize the following special actions:

- A *creation action* that has the form $Crt\_\mathcal{A}'?$ or $Crt\_\mathcal{A}'!$ denotes a message for creation of DLHA $\mathcal{A}'$. $Crt\_\mathcal{A}'? \in Act_{in}$ is an input action, and it represents that $\mathcal{A}'$ has been created. $Crt\_\mathcal{A}'! \in Act_{out}$ is an output action, and represents a request for creating $\mathcal{A}'$.

- A *destruction action* that has the form $Dst\_\mathcal{A}'?$ or $Crt\_\mathcal{A}'!$ denotes a message for a destruction of DLHA $\mathcal{A}'$. $Dst\_\mathcal{A}'? \in Act_{in}$ is an input action that indicates $\mathcal{A}'$ has been destroyed.

- An *enqueue action* that has the form $q!m$ denotes enqueueing of message *m* into a queue *q*. This action is an internal one, that is, $q!m \in Act_{\tau}$.

- A *dequeue action* that has the form $q?m$ denotes dequeueing of message *m* from the top of *q*.

- $T \subseteq L \times \Phi(V) \times Act \times 2^{UPD(V)} \times L$ is a finite set of *transitions*. A constraint $\phi \in \Phi(V)$ is called a *guard condition*.
- $t_0 \in L \times (Act_{in} \cup Act_{\tau}) \times 2^{UPD(V)}$ is an *initial transition*.
- $T_d \subseteq L \times \Phi(V) \times Act_{out}$ is a finite set of *destruction-transitions*.

2.3. Operational Semantics

A state σ of a DLHA $(L, V, Inv, Flow, T, t_0, T_d)$ is defined as

$$\sigma ::= \perp \mid (l, v),$$

where $l \in L$ is a location, $v : V \to \mathbb{R}$ is an assignment called *evaluation* of variables, and $\perp$ denotes an *undefined value*.

We define the semantics $\mathcal{M}$ of the DLHA by $(\Sigma, \Rightarrow, \sigma_0)$, where

- Σ is a set of states.
- $\Rightarrow$ is a set of *time transitions* and *discrete transitions*.
- σ_0 is the initial state.

The following rules define time and discrete transitions:

Definition 4 (Time transition of a DLHA). *For any* $\delta \in \mathbb{R}_{\geq 0}$,

- $\perp \Rightarrow_\delta \perp$
- $(l, v) \Rightarrow_\delta (l, v')$ if $v' = v + \delta \cdot Flow(l) \in Inv(l)$

where $v' = v + \delta \cdot Flow(l)$ denotes an evaluation such that $\forall x \in V. v'(x) = v(x) + \delta \cdot \dot{x} \wedge Flow(l)$, and $v' \in Inv(l)$ denotes that $v'(x)$ satisfies the constraint $Inv(l)$ for any $x \in V$.

Definition 5 (Discrete transition of a DLHA). *For an evaluation* v *and update expressions* $\lambda \in 2^{UPD(V)}$, $v[\lambda]$ *denotes an evaluation updated by* λ, *that is, for any* $x \in V$,

$$v[\lambda](x) = \begin{cases} c & (x := c \in \lambda) \\ v(x) + c & (x := c \notin \lambda, x := x + c \in \lambda) \\ v(x) & (\text{otherwise}) \end{cases}$$

- For any transition $(l, \phi, a, \lambda, l') \in T$, $(l, v) \Rightarrow_a (l, v[\lambda])$ if $v \in \phi$ and $v[\lambda] \in Inv(l')$.
- (*Creation of a DLHA*) For the initial transition $t_0 = (l_0, a_0, \lambda_0)$, $\perp \Rightarrow_{a_0} (l_0, \vec{0}[\lambda_0])$, where $\vec{0}$ is an evaluation such that $\forall x \in V. \vec{0}(x) = 0$.
- (*Destruction of a DLHA*) For any destruction-transition $(l, \phi, a) \in T_d$, $(l, v) \Rightarrow_a \perp$ if $v \in \phi$

For the initial transition (l_0, a_0, λ_0), the initial state σ_0 is defined as

$$\sigma_0 = \begin{cases} \perp & (a_0 \in Act_{in}) \\ (l_0, \vec{0}[\lambda_0]) & (\text{otherwise}). \end{cases}$$

3. Dynamically Reconfigurable Systems

To describe an asynchronous communication among DLHAs in a dynamically reconfigurable system (DRS), we use a queue (*unbounded* FIFO buffer) as a model of the communication channel. We assume that the system performs lossless transmission, so we can let the queue be unbounded.

3.1. Syntax of DRS

A dynamically reconfigurable system (DRS) $\mathcal{S}$ is defined by a tuple $(A, \mathcal{Q})$ consisting of a finite set $A = \{\mathcal{A}_1, \cdots, \mathcal{A}_{|A|}\}$ of DLHAs and a finite set $\mathcal{Q} = \{q_1, \cdots, q_{|\mathcal{Q}|}\}$ of queues.

3.2. Semantics of DRS

A state s of a DRS $\mathcal{S} = (A, \mathcal{Q})$ is a tuple $\langle \vec{\sigma}, \vec{w}_{\mathcal{Q}} \rangle$, where

- $\vec{\sigma} \in \Sigma_1 \times \cdots \times \Sigma_{|A|}$ is a vector of the states of DLHAs.
- $\vec{w}_{\mathcal{Q}} \in M_1^* \times \cdots \times M_{|\mathcal{Q}|}^*$ is a vector of the content of the queues, where each M_i is the set of all messages that can be stored in queue q_i.

Definition 6 (Time Transition of a DRS). *For an arbitrary* $\delta \in \mathbb{R}_{\geq 0}$, *the time transition is defined as*

$$\langle \vec{\sigma}, \vec{w}_{\mathcal{Q}} \rangle \rightarrow_{\delta} \langle \vec{\sigma}', \vec{w}_{\mathcal{Q}} \rangle \Leftarrow \forall i. \sigma_i \Rightarrow_{\delta} \sigma_i.$$

Definition 7 (Discrete Transition of a DRS). *Let* $\vec{\sigma}, \vec{\sigma}', \vec{w}_{\mathcal{Q}}$ *and* $\vec{w}'_{\mathcal{Q}}$ *be* $\vec{\sigma} = (\sigma_1, \cdots, \sigma_{|A|})$, $\vec{\sigma}' = (\sigma'_1, \cdots, \sigma'_{|A|})$, $\vec{w}_{\mathcal{Q}} = (w_1, \cdots, w_{|\mathcal{Q}|})$, *and* $\vec{w}'_{\mathcal{Q}} = (w'_1, \cdots, w'_{|\mathcal{Q}|})$.

- For any output action $a!$, $\langle \vec{\sigma}, \vec{w}_{\mathcal{Q}} \rangle \rightarrow_a \langle \vec{\sigma}', \vec{w}_{\mathcal{Q}} \rangle$

$$\text{if } \exists i. \sigma_i \Rightarrow_{a!} \sigma_{i'} \wedge \left(\forall j \neq i. \sigma_j \Rightarrow_{a?} \sigma_j \vee \left(\left(\neg \exists \sigma_{j'}. \sigma_j \Rightarrow_{a?} \sigma_{j'} \right) \wedge \sigma_j = \sigma_{j'} \right) \right).$$

An output action is broadcasted to all DLHAs, and a DLHA receiving the action moves by synchronization if the guard condition holds in the state.

- For an internal action a_τ,

  - in the case of $a_\tau = q_k!w$, $\langle \vec{\sigma}, \vec{w}_{\mathcal{Q}} \rangle \rightarrow_{q_k!w} \langle \sigma', \vec{w}'_{\mathcal{Q}} \rangle$,

$$\text{if } \left(\exists i. \sigma_i \Rightarrow_{q_k!w} \sigma_{i'} \wedge \forall j \neq i. \sigma_j = \sigma_{j'} \right) \wedge w_{k'} = w_k w \wedge \forall l \neq k. w_k = w_{k'},$$

  - while in the case of $a_\tau = q_k?w$, $\langle \vec{\sigma}, \vec{w}_{\mathcal{Q}} \rangle \rightarrow_{q_k?w} \langle \vec{\sigma}', \vec{w}'_{\mathcal{Q}} \rangle$,

$$\text{if } \left(\exists i. \sigma_i \Rightarrow_{q_k?w} \sigma_{i'} \wedge \forall j \neq i. \sigma_j = \sigma'_j \right) \wedge w_k = w w'_k \wedge \forall l \neq k. w_l = w'_l,$$

  - otherwise, $\langle \vec{\sigma}, \vec{w}_{\mathcal{Q}} \rangle \rightarrow_{a_\tau} \langle \vec{\sigma}', \vec{w}_{\mathcal{Q}} \rangle$ if $\exists i. \sigma_i \Rightarrow_{a_\tau} \sigma_{i'} \wedge \forall j \neq i. \sigma_j = \sigma'_j$.

A *run* (*or path*) ρ of the system $\mathcal{S}$ is the following finite (or infinite) sequence of states.

$$\rho : s_0 \rightarrow_{a_0}^{\delta_0} s_1 \rightarrow_{a_1}^{\delta_1} \cdots \rightarrow_{a_{i-1}}^{\delta_{i-1}} s_i \rightarrow_{a_i}^{\delta_i} \cdots$$

where $\rightarrow_{a_i}^{\delta_i}$ between s_i and s_{i+1} is defined as

$$s_i \rightarrow_{a_i}^{\delta_i} s_{i+1} \Leftarrow \exists s'_i. s_i \rightarrow_{\delta_i} s'_i \wedge s'_i \rightarrow_{a_i} s_{i+1}.$$

The initial state s_0 of a dynamically reconfigurable system is $\langle (\sigma_{01}, \cdots, \sigma_{0|A|}), (w_{01}, \cdots, w_{0|\mathcal{Q}|}) \rangle$ where each σ_{0i} is the initial state of DLHA $\mathcal{A}_i$ and each w_{0j} is empty; that is, $\forall j. w_{0j} = \varepsilon$.

Example 1 (DLHA and DRS). *A DLHA is represented by a directed graph, where each node represents a location and each edge represents a transition.* Figure 1 *shows a dynamically reconfigurable system* $\mathcal{S}$ *consisting of three DLHAs and one queue.*

$$\mathcal{A}_1 = (L_1, V_1, Inv_1, Flow_1, Act_1, T_1, t_{01}, T_{d1}),$$

$$\mathcal{A}_2 = (L_2, V_2, Inv_2, Flow_2, Act_2, T_2, t_{02}, T_{d2}),$$

$$\mathcal{A}_3 = (L_3, V_3, Inv_3, Flow_3, Act_3, T_3, t_{03}, T_{d3}),$$

$$\mathcal{S} = (\{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3\}, \{q\})$$

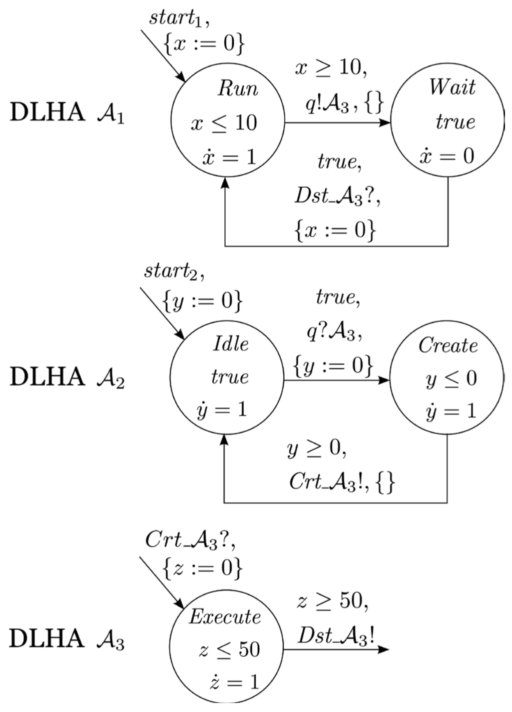

Figure 1. Example of DRS consisting of three DLHAs and one queue.

where

$$L_1 = \{Run, Wait\}$$

$$V_1 = \{x\}$$

$$Inv_1 = \{Run \mapsto x \leq 10, Wait \mapsto true\}$$

$$Flow_1 = \{Run \mapsto \dot{x} = 1, Wait \mapsto \dot{x} = 0\}$$

$$Act_1 = \{Dst\_\mathcal{A}_3?, start_1, q!\mathcal{A}_3\}$$

$$T_1 = \{(Run, x \geq 10, q!\mathcal{A}_3, \{\}, Wait), (Wait, true, Dst\_\mathcal{A}_3?, \{x := 0\}, Run)\}$$

$$t_{01} = (Run, start_1, \{x := 0\})$$

$$T_{d1} = \{\}$$

$$L_2 = \{Idle, Create\}$$

$$V_2 = \{y\}$$

$$Inv_2 = \{Idle \mapsto true, Create \mapsto y \leq 0\}$$

$$Flow_2 = \{Idle \mapsto \dot{y} = 1, Create \mapsto \dot{y} = 1\}$$

$$Act_2 = \{Crt\_\mathcal{A}_3!, start_2, q?\mathcal{A}_3\}$$

$$T_2 = \{(Idle, true, q?\mathcal{A}_3, \{y := 0\}, Create), (Create, y \geq 0, Crt\_\mathcal{A}_3!, \{\}, Idle)\}$$

$$t_{02} = (Idle, start_2, \{y := 0\})$$

$$T_{d2} = \{\}$$

$$L_3 = \{Execute\}$$

$$\begin{aligned}
V_3 &= \{z\} \\
Inv_3 &= \{Execute \mapsto z \le 50\} \\
Flow_3 &= \{Execute \mapsto \dot{z} = 1\} \\
Act_3 &= \{Crt\_\mathcal{A}_3?, Dst\_\mathcal{A}_3!\} \\
T_3 &= \{\ \} \\
t_{03} &= (Execute, Crt\_\mathcal{A}_3?, \{z := 0\}) \\
T_{d3} &= \{(Execute, z \ge 50, Dst\_\mathcal{A}_3!)\}
\end{aligned}$$

This system runs as follows:

1) $\mathcal{A}_1$ requires $\mathcal{A}_3$ to be created from $\mathcal{A}_2$ by enqueueing a message, and it waits for the message to return from $\mathcal{A}_3$.

2) When $\mathcal{A}_2$ receives the message, it creates $\mathcal{A}_3$.

3) After $\mathcal{A}_3$ finishes processing the job, it sends the message to $\mathcal{A}_1$ and is destroyed.

4) This system infinitely repeats steps 1) to 3).

For example, (1) shows a run ρ of this system is shown.

$$\begin{aligned}
&\rho : \langle((Run, x=0),(Idle, y=0),\perp),(\varepsilon)\rangle \\
&\to^{10}_{q!\mathcal{A}_3} \langle((Wait, x=10),(Idle, y=0),\perp),(\mathcal{A}_3)\rangle \\
&\to^{0}_{q?\mathcal{A}_3} \langle((Wait, x=10),(Create, y=0),\perp)),(\varepsilon)\rangle \\
&\to^{0}_{Crt\_\mathcal{A}_3} \langle((Wait, x=10),(Idle, y=0),(Execute, z=0)),(\varepsilon)\rangle \\
&\to^{50}_{Dst\_\mathcal{A}_3} \langle((Run, x=0),(Idle, y=0),\perp),(\varepsilon)\rangle \\
&\to \cdots
\end{aligned} \tag{1}$$

4. Reachability Analysis

4.1. Reachability Problem

We define reachability and the reachability problem for a DRS as follows:

Definition 8 (Reachability). *For a DRS $\mathcal{S} = (A, \mathcal{Q})$ and a location l_t, $\mathcal{S}$ reaches l_t if there exists a path such that*

$$s_0 \to^{\delta_0}_{a_0} \cdots \to^{\delta_{t-1}}_{a_{t-1}} s_t$$

$$\wedge s_t = \langle(\sigma_1, \cdots, \sigma_{|A|}), \vec{w}_Q\rangle, \exists k.loc(\sigma_k) = l_t,$$

where

$$loc(\sigma) = \begin{cases} l & (\sigma = (l, \nu)) \\ \perp(\text{undefined}) & (\sigma = \perp) \end{cases}$$

Definition 9 (Reachability Problem). *Given a DRS $\mathcal{S} = (A, \mathcal{Q})$ and a location l_t, we output "yes" if $\mathcal{S}$ can reach l_t, and "no" otherwise.*

4.2. Reachability Analysis

4.2.1. Convex Polyhedra

Our method introduces *convex polyhedra* for the reachability analysis in accordance with [15].

A polyhedron is convex if it can be defined by a formula which is a conjunction of linear formulae. For a set $V = \{x_1, \cdots, x_n\}$ of variables, a convex polyhedron ζ on V is a *n*-dimensional real space. In particular, we define *true* and *false* as $true = \mathbb{R}^n$ and $false = \varnothing$.

Example 2 (Convex Polyhedron). *The formula* $\exists x_1 . x_1 \geq 5 \wedge x_2 \leq 1 \wedge x_1 - x_2 = 1 \wedge true$ *is a convex polyhedron. From linear formula, the existential quantifier can be eliminated effectively. Therefore, we obtain*

$$\begin{aligned}
&\exists x_1 . x_1 \geq 5 \wedge x_2 \leq 1 \wedge x_1 - x_2 = 1 \wedge true \\
&= \exists x_1 . x_1 \geq 5 \wedge x_2 \leq 1 \wedge x_1 - x_2 = 1 \\
&= x_2 \leq 1 \wedge x_2 \geq 4 \\
&= false.
\end{aligned}$$

4.2.2. Algorithm of Reachability Analysis

We define a state s in the reachability analysis as $(L, \zeta, \vec{w}_Q)$, where

- L is a finite set of locations.
- ζ is a convex polyhedron.
- $\vec{w}_Q$ is a vector of the content of the queues.

Figure 2, Figure 3 and Figure 4 show the algorithm of the reachability analysis.

Figure 2 is an overview of the reachability analysis, and this algorithm is performed using the expanded method of [16] with a set Q of queues. The analysis is performed as follows:

1. Compute an initial state s_0 of the system $\mathcal{S}$ (ll.1-3).

2. Initialize a traversed set *Visit* and a untraversed set *Wait* of states by $\varnothing$ and $\{s_0\}$ (line 4).

3. While *Wait* is not empty, repeat the following process (ll.5-16).

(a) Take a state $(L, \zeta, \vec{w}_Q)$ from *Wait* and remove the state from *Wait* (ll.6-7).

```
Input: a system $\mathcal{S}$ and a target location $l_t$
Output: "yes" or "no"
 1: $L_0 \leftarrow \{l_{0i} \mid t_{0i} = (l_{0i}, a_{0i}, \lambda_{0i}), a_{0i} \neq Crt\_\mathcal{A}_i?\}$
 2: $\lambda_0 \leftarrow \bigcup\{\lambda_{0i} \mid t_{0i} = (l_{0i}, a_{0i}, \lambda_{0i}), a_{0i} \neq Crt\_\mathcal{A}_i?\}$
 3: $s_0 \leftarrow (L_0, \vec{0}[\lambda_0], (\varepsilon, \ldots, \varepsilon))$ /* Compute the initial state */
 4: Visit $\leftarrow \varnothing$, Wait $\leftarrow \{s_0\}$ /* Initialize */
 5: while Wait $\neq \varnothing$ do
 6:   $(L, \zeta, \vec{w}_Q) \leftarrow s \in$ Wait
 7:   Wait $\leftarrow$ Wait $\setminus \{(L, \zeta, \vec{w}_Q)\}$
 8:   if $l_t \in L$ then
 9:     return "yes"
10:   end if
11:   if $(L, \zeta, \vec{w}_Q) \notin$ Visit then
12:     Visit $\leftarrow$ Visit $\cup \{(L, \zeta, \vec{w}_Q)\}$
13:     $S_{post} \leftarrow$ Succ$((L, \zeta, \vec{w}_Q), \mathcal{S})$ /* Compute the set of post-states */
14:     Wait $\leftarrow$ Wait $\cup S_{post}$
15:   end if
16: end while
17: return "no"
```

Figure 2. Reachability analysis.

```
Input: a state $(L, \zeta, \vec{w}_Q)$ and the system $\mathcal{S}$
Output: the set $S_{post}$ of post-states
1: $T_N \leftarrow \bigcup_{i=1}^{|A|}\{(l, \phi_g, a, \lambda, l') \in T_i \mid l \in L\}$ /* Set of outgoing transitions */
2: $T_D \leftarrow \bigcup_{i=1}^{|A|}\{(l, \phi_g, a) \in T_{di} \mid l \in L\}$ /* Set of outgoing destruction-transitions */
3: $S_{post} \leftarrow \varnothing$, $T_{post} \leftarrow T_N \cup T_D$
4: $\zeta_\delta \leftarrow \mathrm{Tsucc}(L, \zeta) \wedge \bigwedge_{l_p \in L} Inv_s(l_p)$ /* Convex polyhedron for the time transition */
5: for all $t \in T_{post}$ do
6:   if $t = (l, \phi_g, a, \lambda, l')$ then
7:     if $a$ is an internal action then
8:       $L' \leftarrow (L \setminus \{l\}) \cup \{l'\}$ /* Locations of the post-state */
9:       $\zeta' \leftarrow (\zeta_\delta \wedge \phi_g)[\lambda] \wedge \zeta'_i \wedge \bigwedge_{l'_p \in L'} Inv_s(l'_p)$ /* The convex polyhedron of the post-states */
10:      if $\zeta' \neq false$ then
11:        if $a$ is a enqueue action $q_k!w$ then
12:          $(w_1, \ldots, w_{|Q|}) \leftarrow \vec{w}_Q$
13:          $w'_k \leftarrow w_k w$ /* Enqueue the message into $q_k$ */
14:          $\vec{w}'_Q \leftarrow (w_1, \ldots, w_{k-1}, w'_k, w_{k+1}, \ldots, w_{|Q|})$
15:          $S_{post} \leftarrow S_{post} \cup \{(L', \zeta', \vec{w}'_Q)\}$
16:        else if $a$ is a dequeue action $q_k?w$ then
17:          if $w_k = w w'_k$ then
18:            $\vec{w}'_Q \leftarrow (w_1, \ldots, w_{k-1}, w'_k, w_{k+1}, \ldots, w_{|Q|})$ /* Dequeue the message from $q_k$ */
19:            $S_{post} \leftarrow S_{post} \cup \{(L', \zeta', \vec{w}'_Q)\}$
20:          end if
21:        else
22:          $S_{post} \leftarrow S_{post} \cup \{(L', \zeta', \vec{w}_Q)\}$ /* For other internal action */
23:        end if
24:      end if
25:    else if $a$ is an output action $a_l!$ then
26:      $S_{post} \leftarrow S_{post} \cup \mathrm{Syncs}((L, \zeta, \vec{w}_Q), t, \mathcal{S})$ /* Compute the set of states using the synchronous transitions */
27:    end if
28:  else
29:    $S_{post} \leftarrow S_{post} \cup \mathrm{Syncs}((L, \zeta, \vec{w}_Q), t, \mathcal{S})$ /* Compute the set of states using the destruction-transition */
30:  end if
31: end for
32: return $S_{post}$
```

Figure 3. Subroutine Succ.

(b) If the set L of locations contains the target location, return "yes" and terminate (ll.8-10).

(c) If the state has not been traversed yet ($(L, \zeta, \vec{w}_Q) \notin Visit$) (line 11),

i. add the state into $Visit$ (line 12),

ii. compute the set S_{post} of successors by using the subroutine $Succ$ (line 13), and

iii. add all components of S_{post} to $Wait$ (line 14).

Subroutine Succ Figure 3 shows the subroutine $Succ$ to compute the successors of a state. In this algorithm, we make the following assumptions.

$$\mathcal{S} = (A, \mathcal{Q}) = \left(\left\{\mathcal{A}_1, \cdots, \mathcal{A}_{|A|}\right\}, \left\{q_1, \cdots, q_{|\mathcal{Q}|}\right\}\right),$$

$$Inv_s = \bigcup_{k=1}^{|A|} Inv_k,$$

where

$$\mathcal{A}_i = \left(L_i, V_i, Inv_i, Flow_i, Act_i, T_i, t_{0i}, T_{di}\right) \text{ is a DLHA},$$

$$t_{0i} = \left(l_{0i}, a_{0i}, \lambda_{0i}\right).$$

```
Input: a state $(L, \zeta, \vec{w}_Q)$, a transition (or destruction-transition) $t_s = (l, \phi_g, a_l!, \lambda, l') \mid (l, \phi_g, a_l!)$ and the system $\mathcal{S}$
Output: the set $S_{post}$ of post-states
1:  $S_{post} \leftarrow \varnothing$
2:  $\zeta_\delta \leftarrow \mathrm{Tsucc}(L, \zeta) \wedge \bigwedge_{l_p \in L} Inv_s(l_p)$ /* Convex polyhedron for the time transition */
3:  for $i = 1$ to $|A|$ do
4:    $T_{si} \leftarrow \{(l_i, \phi_i, a_i, \lambda_i, l'_i) \in T_i \mid l_i \in (L \setminus \{l\}), a_i = a_l?, \zeta_\delta \wedge \phi_i \neq false\}$ /* Synchronized transitions of $\mathcal{A}_i$ */
5:  end for
6:  $\Delta \leftarrow \prod\{T_{si} \mid T_{si} \neq \varnothing, i \in \{1, \dots, |A|\}\}$ /* Combinations of transitions */
7:  for all $(t_1, \dots, t_n) \in \Delta$ do
8:    $T_{sync} \leftarrow \max_{|\Delta'|} \Delta' \subseteq \{t_1, \dots, t_n\}$ s. t. $\zeta_\delta \wedge \phi \wedge \bigwedge\{\phi_s \mid (l_1, \phi_s, a_l?, \lambda_s, l_2) \in \Delta'\} \neq false$ /* The set of transitions synchronized with $t_s$ */
9:    $L_{pre} \leftarrow \{l\}$, $L_{post} \leftarrow \varnothing$, $\phi \leftarrow \phi_g$, $\lambda_u \leftarrow \varnothing$
10:   for all $t_{in} \in T_{sync}$ do
11:     $(l_i, \phi_i, a_i, \lambda_i, l'_i) \leftarrow t_{in}$
12:     $L_{pre} \leftarrow L_{pre} \cup \{l_i\}$, $L_{post} \leftarrow L_{post} \cup \{l'_i\}$ /* Pre-locations and post-locations of $T_{sync}$ */
13:     $\phi \leftarrow \phi \wedge \phi_i$, $\lambda_u \leftarrow \lambda_u \cup \lambda_i$ /* Guard conditions and update expressions */
14:   end for
15:   if $t_s = (l, \phi_g, a, \lambda, l')$ then
16:     $L_{post} \leftarrow L_{post} \cup \{l'\}$, $\lambda_u \leftarrow \lambda_u \cup \lambda$ /* Add the post-location and the update expressions of $t_s$ */
17:   end if
18:   if $a_l = Crt\_\mathcal{A}_j$ and $L_j \cap L = \varnothing$ then
19:     $L_{post} \leftarrow L_{post} \cup \{l_{0j}\}$, $\lambda_u \leftarrow \lambda_u \cup \lambda_{0i}$ /* If $\mathcal{A}_j$ is not yet created, add the initial location and update expressions */
20:   end if
21:   $L'_T \leftarrow (L \setminus L_{pre}) \cup L_{post}$ /* Locations of the post-state */
22:   $\zeta'_T \leftarrow (\zeta_\delta \wedge \phi)[\lambda_u] \wedge \bigwedge_{l'_p \in L'} Inv_s(l'_p)$ /* The convex polyhedron of the post-state */
23:   if $t_s = (l, \phi_g, Dst\_\mathcal{A}_i!)$ then
24:     $\zeta'_T \leftarrow \exists V_i . \zeta'_T$ /* If $t_s$ is a destruction-transition, free variables for the convex polyhedron. */
25:   end if
26:   if $\zeta'_T \neq false$ then
27:     $S_{post} \leftarrow S_{post} \cup \{(L'_T, \zeta'_T, \vec{w}_Q)\}$ /* If the transition is possible, add the post-state. */
28:   end if
29: end for
30: return $S_{post}$
```

Figure 4. Subroutine Syncs.

Let the initial state of $\mathcal{S}$ be $\left\langle\left(\sigma_{01}, \cdots, \sigma_{0|A|}\right)\vec{w}_{Q0}\right\rangle$; s_0 is $\left(L_0, \zeta_0, \vec{w}_{Q0}\right)$, where

$$\mathrm{zone}(\sigma) = \begin{cases} \nu & (\sigma = (l, \nu)) \\ true & (\sigma = \perp), \end{cases}$$

$$L_0 = \left\{\mathrm{loc}(\sigma_{0i}) \mid \sigma_{0i} \neq \perp, i \in \{1, \cdots, |A|\}\right\},$$

$$\zeta_0 = \bigwedge_{i=0}^{|A|} \mathrm{zone}(\sigma_{0i}).$$

Here, $\mathrm{zone}(\sigma)$ is a function that assigns a convex polyhedron to each state.

$\mathrm{Tsucc}(L, \zeta)$ is a function that returns a convex polyhedron after performing a time transition on a given set L of locations and a convex polyhedron ζ (line 4). We define this function in accordance with [15] as follows:

Let the set of all variables in the system and their derivatives be

$$V_s = \bigcup_{k=1}^{|A|} V_k = \{x_1, \cdots, x_n\} \text{ and } \dot{V}_s = \{\dot{x}_1, \cdots, \dot{x}_n\}.$$

$$\mathrm{Tsucc}(L, \zeta) = \exists x_1, \cdots, x_n \in V_s . \exists \delta \in \mathbb{R}_{\geq 0} . \exists \dot{x}_1, \cdots, \dot{x}_n \in \dot{V}_s .$$

$$\zeta \wedge Flow(L) \wedge \bigwedge_{x \in V_s} x' = x + \delta \dot{x} \text{ and rename } x' \text{ as } x,$$

where

$$Flow_s = \bigcup_{k=1}^{|A|} Flow_k,$$

$$Flow = \bigwedge_{l \in L} Flow_s(l).$$

For a convex polyhedron ζ and a set λ of update expressions, $\zeta[\lambda]$ denotes the convex polyhedron updated by λ for ζ. Let the set of reset variables and set of shifted variables be $V_r = \{x \mid x := c \in \lambda\} = \{x_{r1}, \cdots, x_{rm}\}$ and $V_a = \{x \mid x := x + c \in \lambda\} = \{x_{a1}, \cdots, x_{an}\}$. $\zeta[\lambda]$ can be computed as

$$\zeta[\lambda] = (\exists x_{r1}, \cdots, x_{rm} \in V_r . \zeta_a) \wedge \bigwedge_{x \in V_r} x = m_r(x),$$

where

$$m_r = \{x \mapsto c \mid x := c \in \lambda\},$$

$$m_a = \{x \mapsto c \mid x := x + c \in \lambda\},$$

$$\zeta_a = \exists x_{a1}, \cdots, x_{an} \in V_a . \zeta \wedge \bigwedge_{x \in V_a} x'$$

$$= x + m_a(x) \text{ and rename variables } x' \text{ as } x.$$

Given a state $(L, \zeta, \vec{w}_Q)$ and a system, the successors are computed using the procedure described below.

1. For each transition $(l, \phi, a, \lambda, l')$ (or destruction-transition $(l, \phi, a_l!)$) outgoing from a location $l \in L$, the set S_{post} of post states is computed as follows (ll.5-31):

(a) Compute the convex polyhedron for the time transition (line 4).

$$\zeta_\delta = \text{Tsucc}(L, \zeta) \wedge \bigwedge_{l_p \in L} Inv_s(l_p).$$

(b) If *a* is an internal action, S_{post} is computed as follows:

i. Compute the set of locations (line 8)

$$L' = (L \setminus \{l\}) \cup \{l'\}.$$

ii. Compute the convex polyhedron for the discrete transition (line 9)

$$\zeta' = (\zeta \wedge \phi)[\lambda] \wedge \bigwedge_{l'_p \in L'} Inv_s(l'_p).$$

iii. If *a* is an enqueue action $q_k!w$ (ll.11-15),

$$S_{post} = \begin{cases} \{(L', \zeta', \vec{w}'_Q)\} & (\zeta' \neq false) \\ \varnothing & (\text{otherwise}), \end{cases}$$

where

$$\vec{w}'_Q = (w_1, \cdots, w_{k-1}, w_k \cdot w, w_{k+1}, \cdots, w_{|Q|}).$$

iv. If *a* is a dequeue action $q_k?w$ (ll.16-20),

$$S_{post} = \begin{cases} \{(L', \zeta', \vec{w}'_Q)\} & (\zeta' \neq false, w_k = w \cdot w'_k, \forall j \neq k. w_j = w'_j) \\ \varnothing & (\text{otherwise}). \end{cases}$$

v. If *a* is another internal action (line 22),

$$S_{post} = \begin{cases} \{(L', \zeta', \vec{w}_Q)\} & (\zeta' \neq false) \\ \varnothing & (\text{otherwise}). \end{cases}$$

(c) If *a* is an output action $a_l!$, S_{post} is computed with the subroutine Syncs (line 26 and 29).

(d) If *a* is an input action, $S_{post} = \varnothing$.

Subroutine Syncs Figure 4 shows the subroutine Syncs of Succ to compute successors by using the transition that has an output action. Given a state $(L, \zeta, \vec{w}_Q)$, a transition (or destruction-transition) $t_s = (l, \phi_g, a_l!, \lambda, l')$, and a system $\mathcal{S} = (A, \mathcal{Q})$, a set S_{post} of successors is computed as follows:

1. Initialize S_{post} as $\varnothing$ (line 1).

2. Compute a convex polyhedron ζ_δ for the time transition (line 2).

$$\zeta_\delta = \text{Tsucc}(L, \zeta) \wedge \bigwedge_{l_p \in L} Inv_s(l_p).$$

3. For each $\mathcal{A}_i$ in the system $\mathcal{S}$, compute the set T_{si} of transitions that are outgoing from the state by using an input action $a_l?$ (ll.3-5),

$$T_{si} = \{(l_i, \phi_i, a_l?, \lambda_i, l'_i) \in T_i \mid l_i \in (L \setminus \{l\})\}.$$

4. Compute the set Δ of combinations of T_{si} (line 6).

$$\Delta = \prod \{T_{si} \mid T_{si} \neq \varnothing, i \in \{1, \cdots, |A|\}\}.$$

5. For each combination $T = (t_1, \cdots, t_n) \in \Delta$, the successor $(L'_T, \zeta'_T, \vec{w}_Q)$ is computed as follows (ll.7-29):

(a) Compute the set T_{sync} of transitions (line 9).

$$T_{sync} = \max_{|\Delta'|} \Delta' \subseteq \{t_1, \cdots, t_n\}$$

$$\text{s.t. } \zeta_\delta \wedge \phi \wedge \bigwedge \{\phi_s \mid (l_1, \phi_s, a_l?, \lambda_s, l_2) \in \Delta'\} \neq false.$$

(b) Compute the set L'_T of locations (ll.9-14, line 21).

$$L'_T = (L \setminus L_{pre}) \cup L_{post},$$

where

$$L_{sync} = \{l_1 \mid (l_1, \phi_s, a_l?, \lambda_s, l_2) \in T_{sync}\},$$

$$L'_{sync} = \{l_2 \mid (l_2, \phi_s, a_l?, \lambda_s, l_2) \in T_{sync}\},$$

$$L_{pre} = \{l\} \cup L_{sync}$$

$$L_{post} = \begin{cases} \{l', l_{j0}\} \cup L'_{sync} & (a_l = Crt\_\mathcal{A}_j, L \cap L_j = \varnothing) \\ L'_{sync} & (a_l = Dst\_\mathcal{A}_j) \\ \{l'\} \cup L'_{sync} & (\text{otherwise}). \end{cases}$$

(c) Compute the update expression λ_{sync} (ll.9-14).

$$\lambda_{sync} = \begin{cases} \lambda \cup \lambda_{0j} \cup \lambda_{in} & \left(a_l = Crt\_\mathcal{A}_j, L \cap L_j = \varnothing\right) \\ \lambda_{in} & \left(a_l = Dst\_\mathcal{A}_j\right) \\ \lambda \cup \lambda_{in} & (\text{otherwise}), \end{cases}$$

where

$$\lambda_{in} = \bigcup \left\{\lambda_s \mid \left(l_1, \phi_s, a_l?, \lambda_s, l_2\right) \in T_{sync}\right\}.$$

(d) Compute the conjunction of guard conditions (ll.9-14).

$$\phi_{sync} = \phi \wedge \bigwedge \left\{\phi_s \mid \left(l_1, \phi_s, a_l?, \lambda_s, l_2\right) \in T_{sync}\right\}.$$

(e) Compute the convex polyhedron ζ_T' (ll.22-24).

$$\zeta_T' = \begin{cases} \exists x_{j1}, \cdots, x_{j|V_j|} \in V_j . \zeta' & \left(a_l = Dst\_\mathcal{A}_j\right) \\ \zeta' & (\text{otherwise}), \end{cases}$$

where

$$\zeta' = \left(\zeta_\delta \wedge \phi_{sync}\right)\left[\lambda_{sync}\right] \wedge \bigwedge_{l_p' \in L'} Inv_s\left(l_p'\right).$$

(f) If $\zeta_T' \neq \text{false}$, the successor is added to S_{post} (ll.26-27).

The correctness of this algorithm is implied by Lemma 1 and Lemma 2.

Lemma 1. *If the algorithm terminates and returns "l_t is not reachable", the system $\mathcal{S}$ has the safety property.*

Lemma 2. *If this algorithm terminates and returns "l_t is reachable", the system $\mathcal{S}$ does not hold the safety property.*

By definition, all linear hybrid automata are DLHAs. Our system dynamically changes its structure by sending and receiving messages. However, the messages statically determine the structure, and the system is a linear hybrid automaton with a set of queues. It is basically equivalent to the reachability analysis of a linear hybrid automaton. Therefore, the reachability problem of DRSs is undecidable, and this algorithm might not terminate [16].

Moreover, in some cases, a system will run into an abnormal state in which the length of a queue becomes infinitely long, and the verification procedure does not terminate.

5. Practical Experiment

5.1. Model Checker

We implemented a model checker of DRSs consisting of DLHAs in Java (about 1600 lines of code) by using the LAS, PPL, and QDD external libraries [12] [17]-[19]. For the verification, we input the DLHAs of the system, a *monitor automaton*, and the *error location* to the model checker, and it output "yes (reachable)" or "no (unreachable)" (**Figure 5**). The monitor automaton had a special location (we call it the error location), and checked the system without changing the system's behavior [15]. The monitor automata had to be specified to reach the error location if the system didn't satisfy the properties.

For the specification of the input model, we extended the syntax and semantics of DLHA as follows:

- A transition between locations can have a label *asap* (that means "as soon as possible"). For a transition labeled *asap*, a time transition does not occur just before the discrete transition.
- Each DLHA can have constraints and update expressions for the variables of another DLHA in the same system. That is, for each DLHA, invariants, guard conditions, update expressions and flow conditions can be used by all DLHAs.

For example, Figure 6 shows the input file for checking whether the system in Figure 1 reaches the location *Execute*.

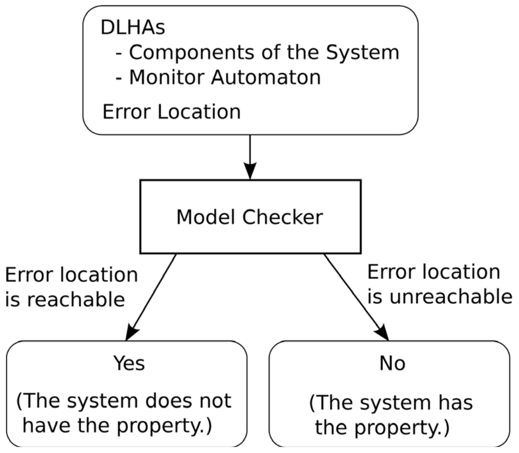
DLHAs
- Components of the System
- Monitor Automaton
Error Location
Model Checker
Error location is reachable
Error location is unreachable
Yes
(The system does not have the property.)
No
(The system has the property.)

Figure 5. Model checker for DRSs.

```
target: Execute
DLHA:
    A1 {
        var: x
        loc Run: x <= 10 [(x,1)]
        loc Wait: true [(x,0)]
        Run -> Wait: x >= 10, q!A3 []
        Wait -> Run: true, DST?A3 [x:=0]
        init: Run, start1 [x:=0]
    }
    A2 {
        var: y
        loc Idle: true [(y,1)]
        loc Create: y <= 0 [(y,1)]
        Idle -> Create: true, q?A3 [y:=0]
        Create -> Idle: y >= 0, CRT!A3 []
        init: Idle, start2 [y:=0]
    }
    A3 {
        var: z
        loc Execute: z <= 50 [(z,1)]
        init: Execute, CRT?A3 [z:=0]
        fin: Execute, z >= 50, DST!A3
    }
```

Figure 6. Example input file: description for checking the reachability of the system in Figure 1.

5.2. Specification of Dynamically Reconfigurable Embedded System

5.2.1. A Cooperative System Including CPU and DRP

We have specified a dynamically reconfigurable embedded system consisting of a CPU and DRP for the model described in our previous research [9]. A DRP is a processor that can execute exclusive processes at the same time by dynamically changing the circuit configuration, and it is used to accelerate CPU computations, for example, in image processing and cipher processing. A DRP has computation resources called *tiles* (or *processing elements*), and it dynamically sets the context of a process if there are enough free tiles. In addition, a DRP can change the operating frequency in accordance with running processes. In this paper, we assume that the number of tiles and the operating frequency for each process have been set in advance and that the operating frequency of the DRP is always the minimum frequency of the running co-tasks.

Figure 7 shows an overview of the system. This system processes jobs submitted from the external environment through the cooperative operation of the CPU and DRP. The CPU Dispatcher creates a task when it receives a call message of the task from the external environment. When a task on the CPU uses the DRP, The CPU Dispatcher sends a message to the DRP Dispatcher. The DRP Dispatcher receives the message asynchronously and creates a *co-task* (it means "cooperative task") in a first-come, first-served manner if there are enough free tiles. Here, we will assume that this system has two tasks and two co-tasks that have the parameters shown in Table 1 & Table 2.

The system, whose components are illustrated in Figure 8, consists of 11 DLHAs and 1 queue. We show part of the state-transition diagram in Figure 9. The external environment consists of EnvA (Figure 10) and EnvB (Figure 11) that periodically create

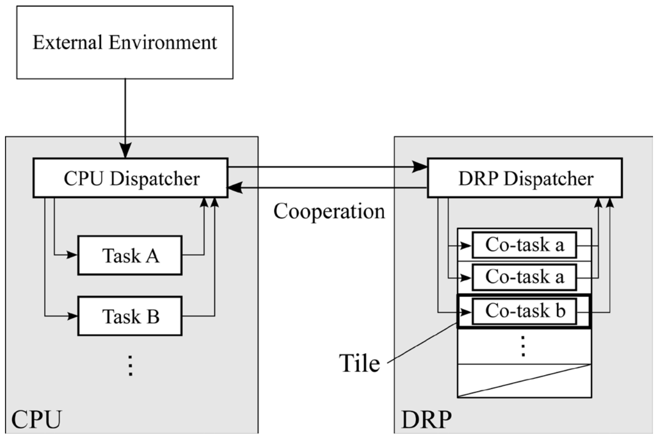

Figure 7. Overview of the CPU-DRP embedded system.

Table 1. Parameters of tasks.

Task	Period	Deadline	Priority	Process
A	70 ms	70 ms	high	20 ms, co-task a0, 10 ms, co-task b0
B	200 ms	200 ms	low	co-task a1, 97 ms

Table 2. Parameters of co-tasks.

co-task	Processing time	Deadline	Tiles	Rate of Frequency
*a*0, *a*1	10 ms	15 ms	2	1
*b*0	5 ms	10 ms	6	1/2

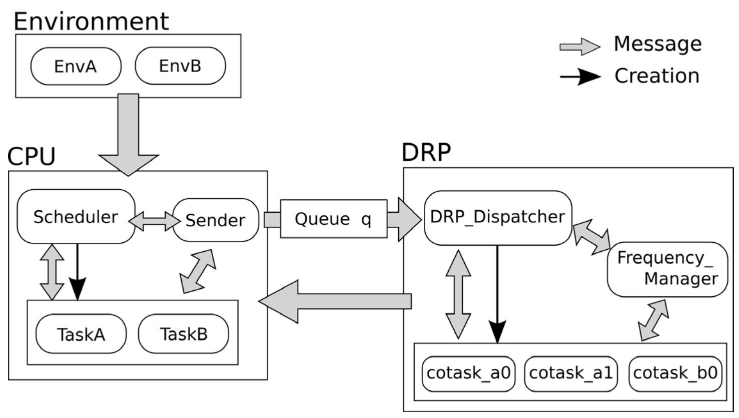

Figure 8. Components of the system.

$$\boxed{\begin{array}{l}\langle\{\mathit{L\_Env\_A}, \mathit{L\_Env\_B}, \mathit{L\_Idle\_TD}, \mathit{SendIdle}, \mathit{L\_Idle\_DD}, \mathit{L\_Freq\_Idle}, \mathit{L\_Idle\_M}\},\\ x_A = 70 \wedge x_B = 200 \wedge \mathit{taskAflag} = 0 \wedge \mathit{taskBflag} = 0 \wedge \mathit{tile} = 8 \wedge \mathit{switch} = -1\\ \wedge\, \mathit{loc\_freq} = 0 \wedge \mathit{counta} = 0 \wedge \mathit{countb} = 0, (\varepsilon)\rangle\end{array}}$$

$$\Big\downarrow \mathit{Crt\_taskA}$$

$$\boxed{\begin{array}{l}\langle\{\mathit{L\_Env\_A}, \mathit{L\_Env\_B}, \mathit{L\_Pri3}, \mathit{SendIdle}, \mathit{L\_Idle\_DD}, \mathit{L\_Freq\_Idle}, \mathit{monitorA}, \mathit{ReadyA}\},\\ x_A = 0 \wedge x_B = 200 \wedge \mathit{taskAflag} = 1 \wedge \mathit{taskBflag} = 0 \wedge \mathit{tile} = 8 \wedge \mathit{switch} = -1\\ \wedge\, \mathit{loc\_freq} = 0 \wedge \mathit{counta} = 0 \wedge \mathit{countb} = 0 \wedge e_A = 0 \wedge r_A = 0, (\varepsilon)\rangle\end{array}}$$

$$\Big\downarrow \mathit{tdPri2}$$

$$\boxed{\begin{array}{l}\langle\{\mathit{L\_Env\_A}, \mathit{L\_Env\_B}, \mathit{L\_Pri2}, \mathit{SendIdle}, \mathit{L\_Idle\_DD}, \mathit{L\_Freq\_Idle}, \mathit{monitorA}, \mathit{ReadyA}\},\\ x_A = 0 \wedge x_B = 200 \wedge \mathit{taskAflag} = 1 \wedge \mathit{taskBflag} = 0 \wedge \mathit{tile} = 8 \wedge \mathit{switch} = -1\\ \wedge\, \mathit{loc\_freq} = 0 \wedge \mathit{counta} = 0 \wedge \mathit{countb} = 0 \wedge e_A = 0 \wedge r_A = 0, (\varepsilon)\rangle\end{array}}$$

$$\Big\downarrow \mathit{Act\_Exec\_A}$$

$$\boxed{\begin{array}{l}\langle\{\mathit{L\_Env\_A}, \mathit{L\_Env\_B}, \mathit{L\_Idle\_TD}, \mathit{SendIdle}, \mathit{L\_Idle\_DD}, \mathit{L\_Freq\_Idle}, \mathit{monitorA}, \mathit{RunA}\},\\ x_A = 0 \wedge x_B = 200 \wedge \mathit{taskAflag} = 1 \wedge \mathit{taskBflag} = 0 \wedge \mathit{tile} = 8 \wedge \mathit{switch} = -1\\ \wedge\, \mathit{loc\_freq} = 0 \wedge \mathit{counta} = 0 \wedge \mathit{countb} = 0 \wedge e_A = 0 \wedge r_A = 0, (\varepsilon)\rangle\rangle\end{array}}$$

$$\Big\downarrow \mathit{Crt\_taskB}$$

$$\boxed{\begin{array}{l}\langle\{\mathit{L\_Env\_A}, \mathit{L\_Env\_B}, \mathit{L\_Pri3}, \mathit{SendIdle}, \mathit{L\_Idle\_DD}, \mathit{L\_Freq\_Idle}, \mathit{monitorA}, \mathit{RunA}, \mathit{ReadyB}\},\\ x_A = 0 \wedge x_B = 0 \wedge \mathit{taskAflag} = 1 \wedge \mathit{taskBflag} = 1 \wedge \mathit{tile} = 8 \wedge \mathit{switch} = -1\\ \wedge\, \mathit{loc\_freq} = 0 \wedge \mathit{counta} = 0 \wedge \mathit{countb} = 0 \wedge e_A = 0 \wedge r_A = 0 \wedge e_B = 0 \wedge r_B = 0 \wedge \mathit{logB} = 0, (\varepsilon)\rangle\end{array}}$$

$$\Big\downarrow \mathit{Act\_Preempt}$$

$$\vdots$$

Figure 9. State-transition diagram of the system.

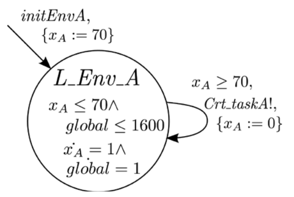

Figure 10. External environment: EnvA.

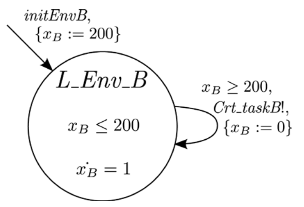

Figure 11. External environment: EnvB.

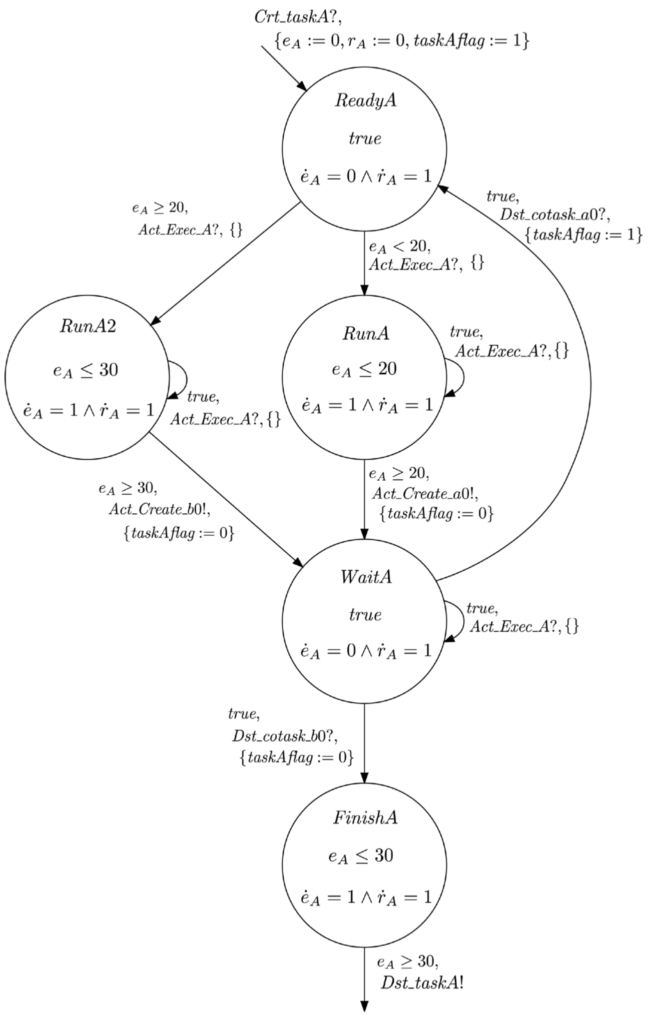

Figure 12. Task: TaskA.

TaskA (Figure 12) and TaskB (Figure 13). That is, EnvA uses *Crt_taskA*! to create TaskA every 70 milliseconds, and EnvB uses *Crt_taskB*! to create TaskB with every 200 milliseconds. The Scheduler (Figure 14) performs scheduling in accordance with the priority and actions for creation and destruction of DLHAs. For example, when TaskA is created by EnvA with *Crt_taskA*! and TaskB is already running, The Scheduler receives *Crt_taskA*? from EnvA and sends *Act_Preempt*! to TaskA and TaskB. Then, *Act_Preempt*! causes TaskA to move to *RunA* and TaskB to move to *WaitB*.

TaskA and TaskB send a message to The Sender if they need a co-task. The Sender (Figure 15) enqueues the message to create a co-task to *q* when it receives a message from tasks. When TaskA sends *Act_Create_a*0! and moves to *RunA* from *WaitA*, The Sender receives *Act_Create_a*0? and enqueues *cotask_a*0 in *q* with *q*! *cotask_a*0.

The DRP_Dispatcher (Figure 16) dequeues a message and creates cotask_a0 (Figure 17), cotask_a1 (Figure 18), and cotask_b0 (Figure 19) if there are enough free tiles. The Frequency_Manager (Figure 20) is a module that manages the operating frequency of the DRP. When a DLHA of a co-task is created, The Frequency_Manager moves to the location that sets the frequency to the minimum value.

5.2.2. Other Cases

We have the parameters of the model in subsection 5.2.1 and conducted experiments with it.

- Modified Tasks: We modified the parameters of the tasks on the CPU as shown in Table 3. Here, the parameters of the co-tasks are the same as those in Table 2.
- Modified co-tasks: We modified the parameters of the co-tasks on the DRP, as shown in Table 4. The parameters of the tasks are the same as those in Table 1.

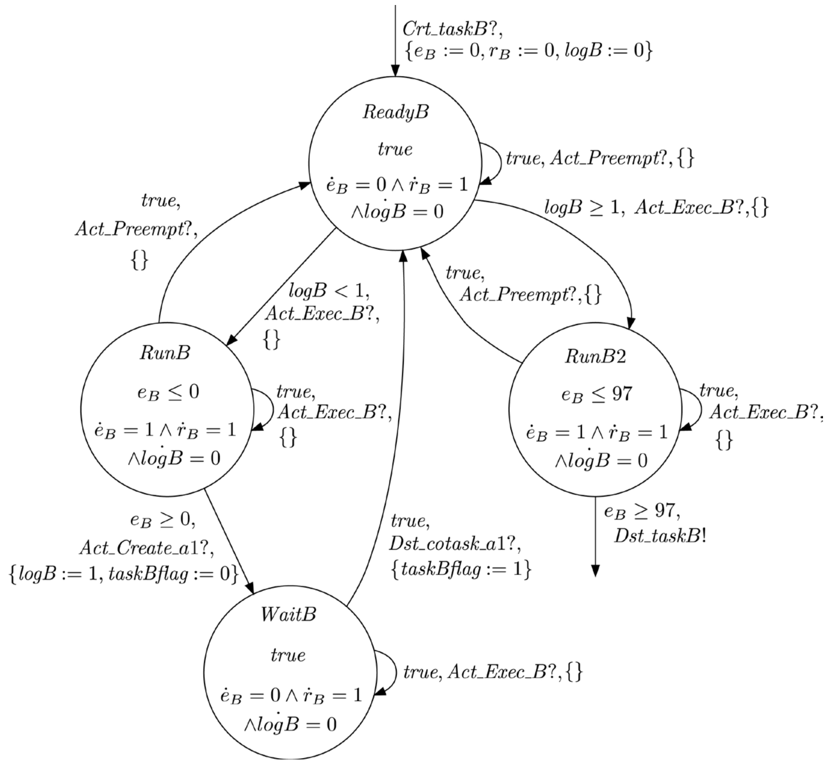
$Crt\_taskB?$, $\{e_B := 0, r_B := 0, logB := 0\}$
ReadyB
true
$\dot{e}_B = 0 \wedge \dot{r}_B = 1 \wedge \dot{logB} = 0$
true, Act_Preempt?, {}
$logB \geq 1$, Act_Exec_B?, {}
true, Act_Preempt?, {}
$logB < 1$, Act_Exec_B?, {}
true, Act_Preempt?, {}
RunB
$e_B \leq 0$
$\dot{e}_B = 1 \wedge \dot{r}_B = 1 \wedge \dot{logB} = 0$
true, Act_Exec_B?, {}
RunB2
$e_B \leq 97$
$\dot{e}_B = 1 \wedge \dot{r}_B = 1 \wedge \dot{logB} = 0$
true, Act_Exec_B?, {}
$e_B \geq 97$, Dst_taskB!
$e_B \geq 0$, Act_Create_a1?, $\{logB := 1, taskBflag := 0\}$
true, Dst_cotask_a1?, $\{taskBflag := 1\}$
WaitB
true
$\dot{e}_B = 0 \wedge \dot{r}_B = 1 \wedge \dot{logB} = 0$
true, Act_Exec_B?, {}

Figure 13. Task: TaskB.

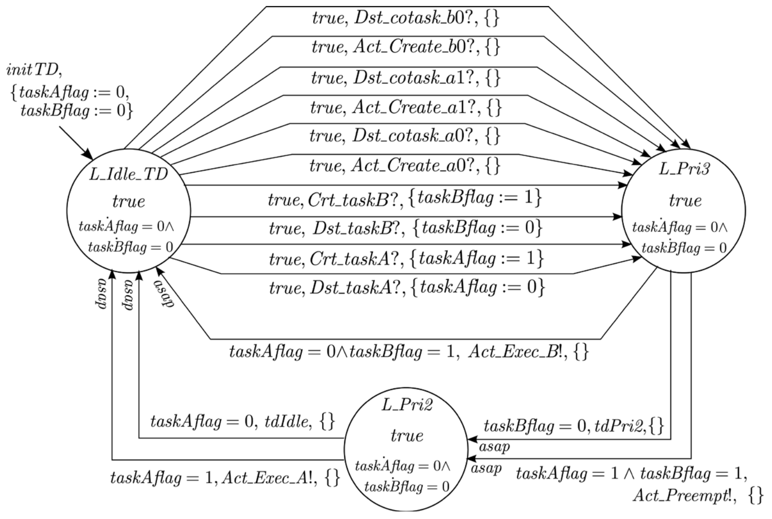
initTD,
{taskAflag := 0, taskBflag := 0}
L_Idle_TD
true
taskAflag = 0∧ taskBflag = 0
L_Pri3
L_Pri2
true, Dst_cotask_b0?, {}
true, Act_Create_b0?, {}
true, Dst_cotask_a1?, {}
true, Act_Create_a1?, {}
true, Dst_cotask_a0?, {}
true, Act_Create_a0?, {}
true, Crt_taskB?, {taskBflag := 1}
true, Dst_taskB?, {taskBflag := 0}
true, Crt_taskA?, {taskAflag := 1}
true, Dst_taskA?, {taskAflag := 0}
taskAflag = 0∧taskBflag = 1, Act_Exec_B!, {}
taskAflag = 0, tdIdle, {}
taskAflag = 1, Act_Exec_A!, {}
taskBflag = 0, tdPri2,{}
taskAflag = 1 ∧ taskBflag = 1, Act_Preempt!, {}
asap

Figure 14. CPU Scheduler: Scheduler.

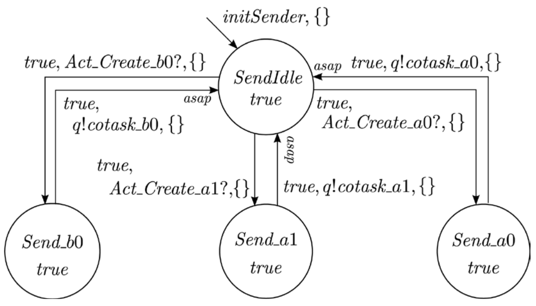

Figure 15. Message sender to DRP: Sender.

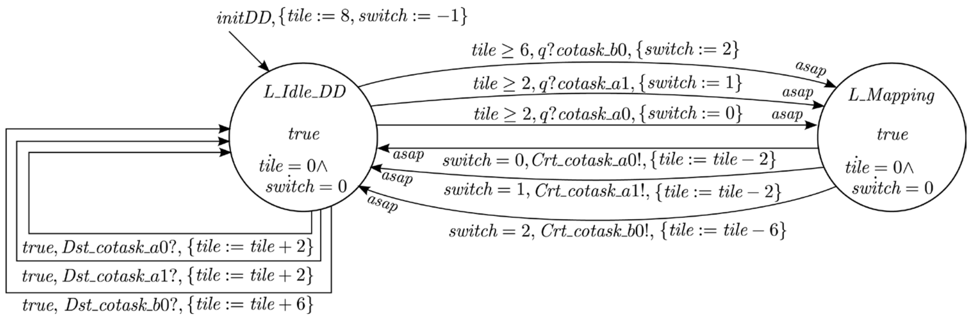
initDD, {tile := 8, switch := −1}
L_Idle_DD
true
tile = 0∧ switch = 0
L_Mapping
tile ≥ 6, q?cotask_b0, {switch := 2}
tile ≥ 2, q?cotask_a1, {switch := 1}
tile ≥ 2, q?cotask_a0, {switch := 0}
switch = 0, Crt_cotask_a0!, {tile := tile − 2}
switch = 1, Crt_cotask_a1!, {tile := tile − 2}
switch = 2, Crt_cotask_b0!, {tile := tile − 6}
true, Dst_cotask_a0?, {tile := tile + 2}
true, Dst_cotask_a1?, {tile := tile + 2}
true, Dst_cotask_b0?, {tile := tile + 6}
asap

Figure 16. DRP_Dispatcher.

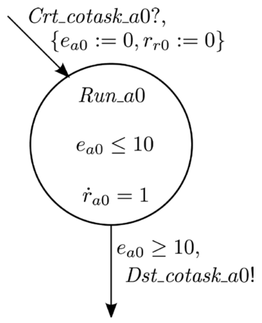

Figure 17. Co-task: cotask_a0.

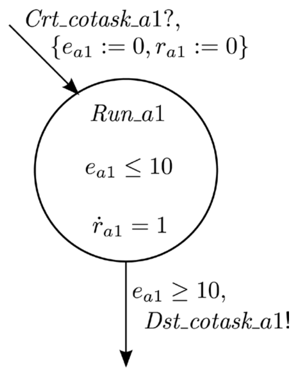
$Crt\_cotask\_a1?$, $\{e_{a1} := 0, r_{a1} := 0\}$
Run_a1
$e_{a1} \leq 10$
$\dot{r}_{a1} = 1$
$e_{a1} \geq 10$, $Dst\_cotask\_a1!$

Figure 18. Co-task: cotask_a1.

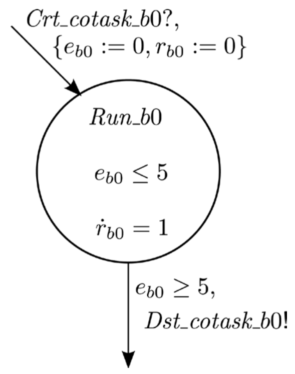
$Crt\_cotask\_b0?$, $\{e_{b0} := 0, r_{b0} := 0\}$
Run_b0
$e_{b0} \leq 5$
$\dot{r}_{b0} = 1$
$e_{b0} \geq 5$, $Dst\_cotask\_b0!$

Figure 19. Co-task: cotask_b0.

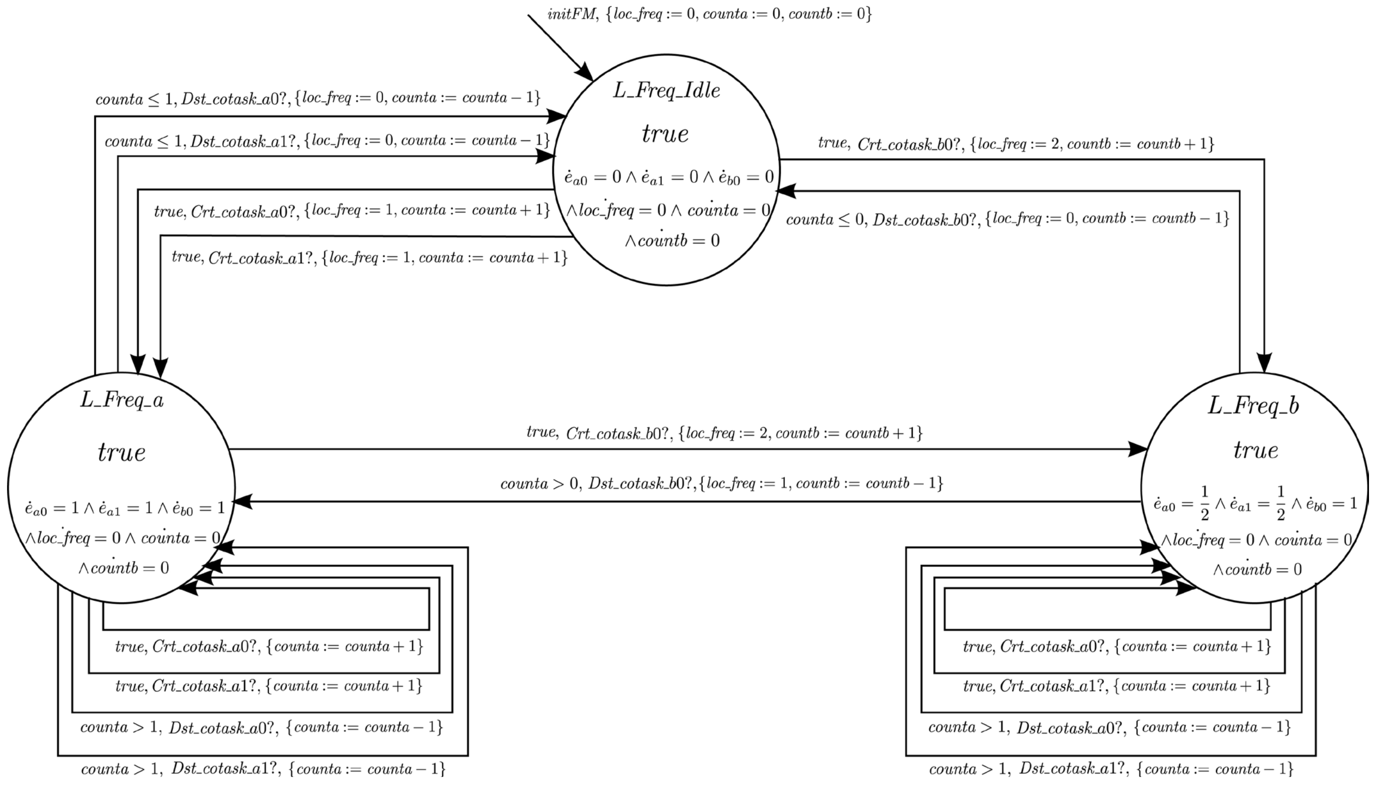
$initFM$, $\{loc\_freq := 0, counta := 0, countb := 0\}$
L_Freq_Idle
$true$
$\dot{e}_{a0} = 0 \wedge \dot{e}_{a1} = 0 \wedge \dot{e}_{b0} = 0$
$\wedge \dot{loc\_freq} = 0 \wedge \dot{counta} = 0$
$\wedge \dot{countb} = 0$
$counta \leq 1, Dst\_cotask\_a0?, \{loc\_freq := 0, counta := counta - 1\}$
$counta \leq 1, Dst\_cotask\_a1?, \{loc\_freq := 0, counta := counta - 1\}$
$true, Crt\_cotask\_a0?, \{loc\_freq := 1, counta := counta + 1\}$
$true, Crt\_cotask\_a1?, \{loc\_freq := 1, counta := counta + 1\}$
$true, Crt\_cotask\_b0?, \{loc\_freq := 2, countb := countb + 1\}$
$counta \leq 0, Dst\_cotask\_b0?, \{loc\_freq := 0, countb := countb - 1\}$
L_Freq_a
$true$
$\dot{e}_{a0} = 1 \wedge \dot{e}_{a1} = 1 \wedge \dot{e}_{b0} = 1$
$\wedge \dot{loc\_freq} = 0 \wedge \dot{counta} = 0$
$\wedge \dot{countb} = 0$
$true, Crt\_cotask\_b0?, \{loc\_freq := 2, countb := countb + 1\}$
$counta > 0, Dst\_cotask\_b0?, \{loc\_freq := 1, countb := countb - 1\}$
L_Freq_b
$true$
$\dot{e}_{a0} = \frac{1}{2} \wedge \dot{e}_{a1} = \frac{1}{2} \wedge \dot{e}_{b0} = 1$
$\wedge \dot{loc\_freq} = 0 \wedge \dot{counta} = 0$
$\wedge \dot{countb} = 0$
$true, Crt\_cotask\_a0?, \{counta := counta + 1\}$
$true, Crt\_cotask\_a1?, \{counta := counta + 1\}$
$counta > 1, Dst\_cotask\_a0?, \{counta := counta - 1\}$
$counta > 1, Dst\_cotask\_a1?, \{counta := counta - 1\}$
$true, Crt\_cotask\_a0?, \{counta := counta + 1\}$
$true, Crt\_cotask\_a1?, \{counta := counta + 1\}$
$counta > 1, Dst\_cotask\_a0?, \{counta := counta - 1\}$
$counta > 1, Dst\_cotask\_a1?, \{counta := counta - 1\}$

Figure 20. Frequency_Manager.

Table 3. Modified parameters of tasks.

Task	Period	Deadline	Priority	Process
A	90 ms	80 ms	high	20 ms, co-task b0,
				20 ms, co-task a0
B	200 ms	150 ms	low	co-task a1, 70 ms

Table 4. Modified parameters of co-tasks.

co-task	Processing time	Deadline	Tiles	Rate of Frequency
*a*0, *a*1	5 ms	10 ms	4	1
*b*0	10 ms	20 ms	5	1/3

5.3. Verification Experiment

We verified that the embedded systems described in subsection 5.2 provide the following properties by using monitor automata (Figures 21-25). The verification experiment was performed on a machine with an Intel (R) Core (TM) i7-3770 (3.40 GHz) CPU and 16 GB RAM running Gentoo Linux (3.10.25-gentoo).

The experimental results shown in Table 5 indicate that the modified tasks cases and the modified co-tasks cases were verified with less computation resources (memory and time) than were used by the original model. This reduction is likely due to the following reasons:

- Regarding the schedulability of the modified tasks model, the processing time is shorter than that of the original model since the verification terminates if a counterexample is found.
- In the cases of the modified co-tasks, the most obvious explanation is that the state-space is smaller than that of the original model since the number of branches in the search tree (*i.e.* nondeterministic transitions in this system) is reduced by changing the start timings of the tasks and co-tasks with the parameters.
- In cases other than those of the modified tasks, it is considered that the state-space is smaller than that of the original model because this system is designed to stop processing when a task exceeds its deadline.

5.3.1. Schedulability

Here, schedulability is a property in which each task of the system finishes before its deadline. Let E_A be the total processing time and D_A be the deadline in task A (Figure 13); the remaining processing time is represented as $E_A - e_A$, and the remaining time till the deadline is represented as $D_A - r_A$. Therefore, the monitor automaton moves the error location if the task A is created and it satisfies the condition $E_A - e_A > D_A - r_A$ (Figure 21). In the case of Table 1, $E_A - e_A > D_A - r_A \Leftrightarrow 30 - e_A > 70 - r_A \Leftrightarrow r_A - e_A > 40$ since $E_A = 30$ and $D_A = 70$. Similarly, the condition for task B is $r_B - e_B > 103$.

5.3.2. Creation of Co-Tasks

In the embedded system, each co-task must be created before the remaining time in the task calling it reaches its deadline. When the message *create_a*0 is received from task A, the monitor automaton starts counting time for co-task *a*0. If the waiting time exceeds the deadline of task A before it receives the message *Crt_cotask_a*0, the monitor moves to error location. Figure 22 shows The monitor automaton for the case of Table 1 for co-task *a*0. Monitor automata for co-tasks *a*1 and *b*0 can be similarly described.

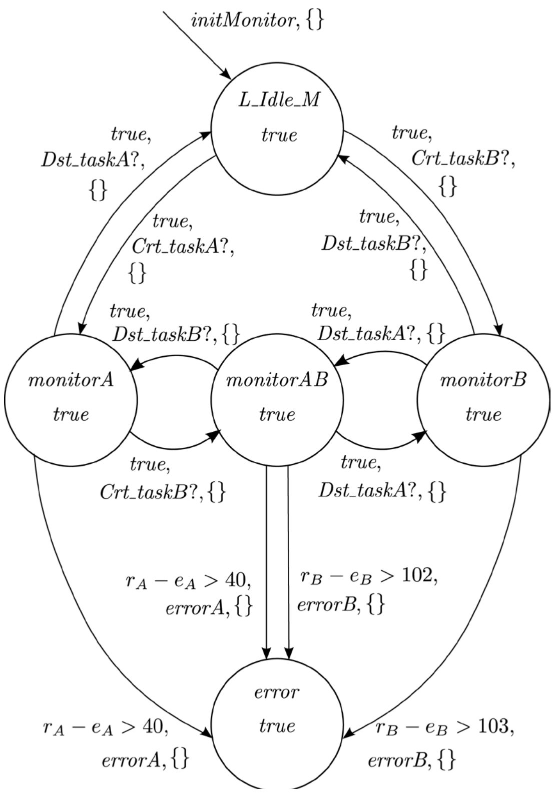

Figure 21. Monitor automaton for checking schedulability.

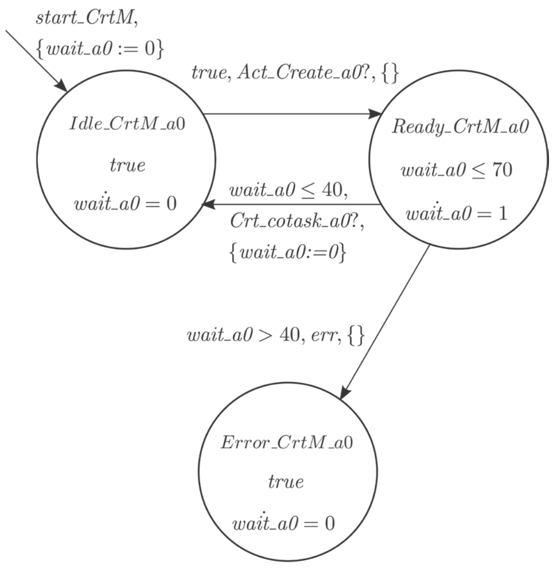

Figure 22. Monitor automaton for checking creation of co-task *a*0.

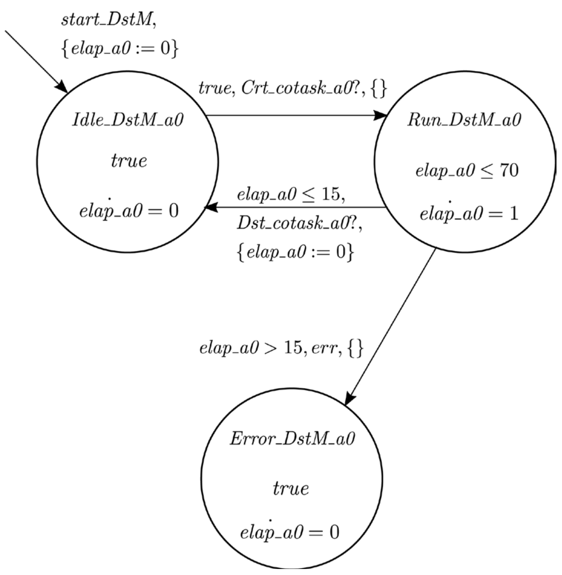

Figure 23. Monitor automaton for checking destruction of co-task *a*0.

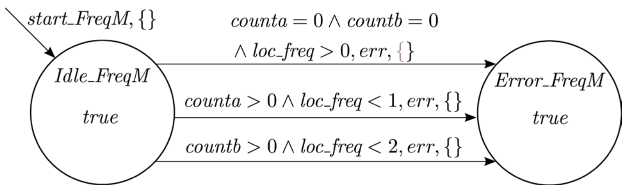
$start\_FreqM, \{\}$
$Idle\_FreqM$
$true$
$counta = 0 \wedge countb = 0$
$\wedge loc\_freq > 0, err, \{\}$
$counta > 0 \wedge loc\_freq < 1, err, \{\}$
$countb > 0 \wedge loc\_freq < 2, err, \{\}$
$Error\_FreqM$
$true$

Figure 24. Monitor automaton for checking frequency management.

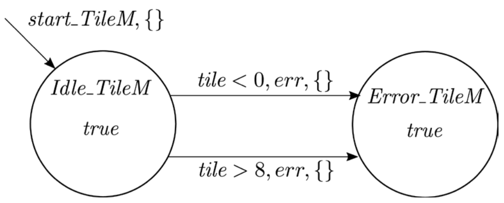

Figure 25. Monitor automaton for checking tile management.

5.3.3. Destruction of Co-Tasks

Each co-task must be destroyed before the waiting time reaches its deadline. For the co-task *a*0, when the message Crt_cotask_a0 is received from the dispatcher DRP_Dispatcher, the monitor automaton checks the message Dst_cotask_a0. Figure 23 shows the monitor automaton for the case of Table 2.

5.3.4. Frequency Management

Creating or destroying a co-task, the DRP changes the operating frequency corresponding to the co-tasks being processed. Since this system requires that the frequency is always at the minimum value, the monitor checks whether the frequency manager (Frequency_Manager) moves to the correct location when it receives a message for creating a co-task. For example, when co-task *a*0 and co-task *b*0 are running on the DRP, Frequency_Manager must be at location L_Freq_b. Figure 24 show the monitor automaton for the case of Table 2.

Table 5. Experimental results.

Model	Property	Satisfiability	Memory [MB]	Time [sec]	The number of states
Original:	Schedulability	yes	168	180	1220
	Creation of co-tasks	yes	92	315	1220
	Destruction of co-tasks	yes	154	233	1220
	Frequency Management	yes	173	265	1220
	Tile Management	yes	167	234	1220
Modified tasks:	Schedulability	no	105	10.2	91
	Creation of co-tasks	yes	117	145	771
	Destruction of co-tasks	yes	82	151	771
	Frequency Management	yes	197	115	771
	Tile Management	yes	135	107	771
Modified co-tasks:	Schedulability	yes	83	141	768
	Creation of co-tasks	yes	85	183	768
	Destruction of co-tasks	yes	86	191	768
	Frequency Management	yes	104	141	768
	Tile Management	yes	119	134	768

5.3.5. Tile Management

When the DRP receives a message for creating of a co-task and the number of free tiles is enough to process it, the dispatcher creates the co-task. The dispatcher then updates the number of used tiles. The monitor automaton checks whether the number *tiles* in DRP_Dispatcher is always between 0 and the maximum number, 8 in this case (Figure 25).

6. Conclusion and Future Work

We proposed a dynamic linear hybrid automaton (DLHA) as a specification language for dynamically reconfigurable systems. We also devised an algorithm for reachability analysis and developed a model checker for verifying the system. Our future research will focus on a more effective method of verification, for example, model checking with CEGAR (Counterexample-guided abstraction refinement) and bounded model checking based on SMT (Satisfiability modulo theories) [20] [21].

References

[1] Agirre, A., Parra, J., Armentia, A., Estévez, E. and Marcos, M. (2016) QoS Aware Middleware Support for Dynamically Reconfigurable Component Based IoT Applications. International Journal of Distributed Sensor Networks, **2016**, Article ID: 2702789. http://dx.doi.org/10.1155/2016/2702789

[2] Garcia, P., Compton, K., Schulte, M., Blem, E. and Fu, W. (2006) An Overview of Reconfigurable Hardware in Embedded Systems. *EURASIP Journal on Embedded Systems*, **2006**

Article ID: 056320. http://dx.doi.org/10.1186/1687-3963-2006-056320

[3] Lockwood, J.W., Moscola, J., Kulig, M., Reddick, D. and Brooks, T. (2003) Internet Worm and Virus Protection in Dynamically Reconfigurable Hardware. *International Conferences on Military and Aerospace Programmable Logic Device* (*MAPLD*), Washington DC, 9-11 September 2003, E10.

[4] Motomura, M., Fujii, T., Furuta, K., Anjo, K., Yabe, Y., Togawa, K., Yamada, J., Izawa, Y. and Sasaki, R. (2005) New Generation Microprocessor Architecture (2) Dynamically Reconfigurable Processor (DRP). *IPSJ Magazine*, **46**, 1259-1265.

[5] Amano, H., Adachi, Y., Tsutsumi, S., Ishikawa, K., *et al.* (2006) A Context Dependent Clock Control Mechanism for Dynamically Reconfigurable Processors. *International Conference on Field Programmable Logic and Applications*, **104**, 575-580. http://dx.doi.org/10.1109/fpl.2006.311269

[6] Vatankhahghadim, A., Song, W. and Sheikholeslami, A. (2015) A Variation-Tolerant MRAM-Backed-SRAM Cell for a Nonvolatile Dynamically Reconfigurable FPGA. *IEEE Transactions on Circuits and Systems II*: *Express Briefs*, **62**, 573-577. http://dx.doi.org/10.1109/TCSII.2015.2407711

[7] Alur, R., Courcoubetis, C., Henzinger, T.A. and Ho, P. (1993) Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. *Lecture Notes in Computer Science*, **736**, 209-229. http://dx.doi.org/10.1007/3-540-57318-6_30

[8] Varshavsky, V. and Marakhovsky, V. (2002) GALA (Globally Asynchronous-Locally Arbitrary) Design. *Lecture Notes in Computer Science*, **2549**, 61-107. http://dx.doi.org/10.1007/3-540-36190-1_3

[9] Minami, S., Takinai, S., Sekoguchi, S., Nakai, Y. and Yamane, S. (2011) Modeling, Specification and Model Checking of Dynamically Reconfigurable Processors. *Computer Software*, **28**, 190-216.

[10] Attie, P.C. and Lynch, N.A. (2001) Dynamic Input/Output Automata, a Formal Model for Dynamic Systems. *Proceedings of the* 20*th Annual ACM Symposium on Principles of Distributed Computing* (*PODC*), **2154**, 314-316. http://dx.doi.org/10.1145/383962.384051

[11] Yamada, H., Nakai, Y. and Yamane, S. (2013) Proposal of Specification Language and Verification Experiment for Dynamically Reconfigurable System. *Information Processing Society of Japan*, **6**, 1-19.

[12] Boigelot, B. and Godefroid, P. (1999) Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs. *Formal Methods in System Design*, **14**, 237-255. http://dx.doi.org/10.1023/A:1008719024240

[13] Bouajjani, A., Esparza, J. and Maler, O. (1997) Reachability Analysis of Pushdown Automata: Application to Model Checking. *Lecture Notes in Computer Science*, **1243**, 135-150. http://dx.doi.org/10.1007/3-540-63141-0_10

[14] Bouajjani, A. and Habermehl, P. (1997) Symbolic Reachability Analysis of FIFO-Channel Systems with Nonregular Sets of Configurations. *Lecture Notes in Computer Science*, **1256**, 560-570. http://dx.doi.org/10.1007/3-540-63165-8_211

[15] Henzinger, T.A., Ho, P. and Wong-Toi, H. (1997) HyTech: A Model Checker for Hybrid Systems. 9*th International Conference on Computer Aided Verification*, Haifa, 22–25 June 1997, 460-463. http://dx.doi.org/10.1007/3-540-63166-6_48

[16] Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P., Nicollin, X., Olivero, A., Sifakis, J. and Yovine, S. (1995) The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, **138**, 3-34. http://dx.doi.org/10.1016/0304-3975(94)00202-T

[17] Ono, Y. and Yamane, S. (2011) Computation of Quantifier Elimination of Linear Inequli-

ties of First Order Predicate Logic. *Ieice Technical Report Theoretical Foundations of Computing*, **111**, 55-59.

[18] Bagnara, R., Hill, P.M. and Zaffanella, E. (2008) The Parma Polyhedra Library: Toward a Complete set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming*, **72**, 3-21. http://dx.doi.org/10.1016/j.scico.2007.08.001

[19] Boigelot, B., Godefroid, P., Willems, B. and Wolper, P. (1997) The Power of QDDs (Extended Abstract). *Proceedings of the 4th International Symposium on Static Analysis*, Paris, 8-10 September 1997, 172-186.

[20] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. (2000) Counterexample-Guided Abstraction Refinement. *Proceedings of the 12th International Conference on Computer Aided Verification*, **1855**, 154-169. http://dx.doi.org/10.1007/10722167_15

[21] Nieuwenhuis, R., Oliveras, A. and Tinelli, C. (2005) Abstract DPLL and Abstract DPLL Modulo Theories. *International Conferences on Logic for Programming, Artificial Intelligence and Reasoning*, **3452**, 36-50. http://dx.doi.org/10.1007/978-3-540-32275-7_3