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

Ryo Yanase¹, Tatsunori Sakai¹, Makoto Sakai¹, Satoshi Yamane²

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

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 reacha-
bility 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, HYTECH [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 HY TECH, 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 \( \phi \) on \( V \) is defined as

\[
\phi ::= \text{true} \mid x \sim e \mid x - y \sim e \mid \phi_1 \land \phi_2,
\]

where \( x, y \in V \), \( e \in \mathbb{Q} \), \( \phi_1 \) and \( \phi_2 \) are constraints on \( V \), and \( \sim \in \{=;<;>;\leq;\geq\} \).

\( \Phi(V) \) denotes the set of all constraints on \( V \).

**Definition 2 (Flow condition).** Let \( V = \{x_1, \ldots, x_n\} \) be a finite set of (real-valued) variables. A flow condition \( f \) on \( V \) is defined as
\[ f := \dot{x}_i = d_1 \land \cdots \land \dot{x}_n = d_n, \]

where \( d_1, \ldots, d_n \in \mathbb{Q} \). \( F(V) \) 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 \( \text{upd} \) on \( V \) is defined as
\[
\text{upd} := x \leftarrow c \mid x \leftarrow x + c,
\]
where \( x \in V \) and \( c \in \mathbb{Q} \). \( \text{UPD}(V) \) is the set of all update expressions can be written.

### 2.2. Syntax

A dynamic linear hybrid automaton (DLHA) is a tuple \( (L, V, \text{Inv}, \text{Flow}, \text{Act}, T, t_0, T_d) \), where
- \( L \) is a finite set of locations.
- \( V \) is a finite set of (real-valued) variables.
- \( \text{Inv} : L \to \Phi(V) \) is a function that assigns a constraint to each location.
- \( \text{Flow} : L \to F(V) \) is a function that assigns a flow condition to each location.
- \( \text{Act} = \text{Act}_{\text{in}} \cup \text{Act}_{\text{out}} \cup \text{Act}_c \) is a finite set of actions.
  - \( \text{Act}_{\text{in}} \) is a finite set of input actions, and each input action has the form \( a? \). An input action \( m? \) denotes receiving the message \( m \).
  - \( \text{Act}_{\text{out}} \) 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.
  - \( \text{Act}_c \) 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 \( \text{Crt}_A)? \) or \( \text{Crt}_A)! \) denotes a message for creation of DLHA \( A' \). \( \text{Crt}_A)? \in \text{Act}_{\text{in}} \) is an input action, and it represents that \( A' \) has been created. \( \text{Crt}_A)! \in \text{Act}_{\text{out}} \) is an output action, and represents a request for creating \( A' \).
- A **destruction action** that has the form \( \text{Dst}_A)? \) or \( \text{Crt}_A)! \) denotes a message for a destruction of DLHA \( A' \). \( \text{Dst}_A)? \in \text{Act}_{\text{in}} \) is an input action that indicates \( A' \) has been destroyed.
- A **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, \( q!m \in \text{Act}_c \).
- A **dequeue action** that has the form \( q?m \) denotes dequeueing of message \( m \) from the top of \( q \).
- \( T \subseteq L \times \Phi(V) \times \text{Act} \times 2^{\text{PO}(V)} \times L \) is a finite set of transitions. A constraint \( \phi \in \Phi(V) \) is called a guard condition.
- \( t_0 \in L \times (\text{Act}_{\text{in}} \cup \text{Act}_c) \times 2^{\text{PO}(V)} \) is an initial transition.
- \( T_d \subseteq L \times \Phi(V) \times \text{Act}_{\text{out}} \) is a finite set of destruction-transitions.

### 2.3. Operational Semantics

A state \( \sigma \) of a DLHA \( (L, V, \text{Inv}, \text{Flow}, T, t_0, T_d) \) is defined as
\[ \sigma := \perp | (l, \nu), \]

where \( l \in L \) is a location, \( \nu : V \rightarrow \mathbb{R} \) 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

- \( \Sigma \) is a set of states.
- \( \Rightarrow \) is a set of *time transitions* and *discrete transitions*.
- \( \sigma_0 \) is the initial state.

The following rules define time and discrete transitions:

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

- \( \perp \Rightarrow_\delta \perp \)
- \( (l, \nu) \Rightarrow_\delta (l, \nu') \) if \( \nu' = \nu + \delta \cdot \text{Flow}(l) \in \text{Inv}(l) \)

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

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

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

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

For the initial transition \( (l_0, a_0, \lambda_0) \), the initial state \( \sigma_0 \) is defined as

\[
\sigma_0 = \begin{cases} 
   \perp & \text{(} a_0 \in \text{Act}_m \text{)} \\
   (l_0, \tilde{\nu}[\lambda_0]) & \text{(otherwise)}
\end{cases}
\]

### 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) \( S \) is defined by a tuple \( (A, Q) \) consisting of a finite set \( A = \{A_1, \ldots, A_m\} \) of DLHAs and a finite set \( Q = \{q_1, \ldots, q_m\} \) of queues.
3.2. Semantics of DRS

A state \( s \) of a DRS \( S = (A, Q) \) is a tuple \( \langle \vec{\sigma}, \vec{w}_Q \rangle \), where
- \( \vec{\sigma} \in \Sigma_1 \times \cdots \times \Sigma_{|A|} \) is a vector of the states of DLHAs.
- \( \vec{w}_Q \in M^*_1 \times \cdots \times M^*_|A| \) 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 \( \delta \in \mathbb{R}_{>0} \), the time transition is defined as
\[
\langle \vec{\sigma}, \vec{w}_Q \rangle \rightarrow_\delta \langle \vec{\sigma}', \vec{w}_Q' \rangle \iff \forall i. \sigma_i \Rightarrow_\delta \sigma_i .
\]

**Definition 7 (Discrete Transition of a DRS).** Let \( \vec{\sigma}, \vec{\sigma}', \vec{w}_Q \) and \( \vec{w}_Q' \) be
\[
\vec{\sigma} = (\sigma_1, \ldots, \sigma_{|A|}), \quad \vec{\sigma}' = (\sigma'_1, \ldots, \sigma'_{|A|}), \quad \vec{w}_Q = (w_1, \ldots, w_{|Q|}), \quad \text{and} \quad \vec{w}_Q' = (w'_1, \ldots, w'_{|Q|}).
\]

- For any output action \( a ! \), \( \langle \vec{\sigma}, \vec{w}_Q \rangle \rightarrow_{\sigma, \vec{w}_Q} \langle \vec{\sigma}', \vec{w}_Q' \rangle \)
  \[
  \text{if } \exists i. \sigma_i \Rightarrow_\delta \sigma_i \land \left( \forall j \neq i. \sigma_j \Rightarrow_\delta \sigma_j \lor \left( \neg \exists \sigma_j. \sigma_j \Rightarrow_\delta \sigma_j \land \sigma_j = \sigma_j \right) \right).
\]

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 \( a \tau \),
  - in the case of \( a \tau = \text{in} \).
    \[
    \langle \vec{\sigma}, \vec{w}_Q \rangle \rightarrow_{\sigma, \vec{w}_Q} \langle \vec{\sigma}', \vec{w}_Q' \rangle \text{ if } \exists i. \sigma_i \Rightarrow_\delta \sigma_i \land \left( \forall j \neq i. \sigma_j \Rightarrow_\delta \sigma_j \lor (\neg \exists \sigma_j. \sigma_j \Rightarrow_\delta \sigma_j \land \sigma_j = \sigma_j) \right).
    \]
  - while in the case of \( a \tau = \text{out} \).
    \[
    \langle \vec{\sigma}, \vec{w}_Q \rangle \rightarrow_{\sigma, \vec{w}_Q} \langle \vec{\sigma}', \vec{w}_Q' \rangle \text{ if } \exists i. \sigma_i \Rightarrow_\delta \sigma_i \land \left( \forall j \neq i. \sigma_j \Rightarrow_\delta \sigma_j \lor (\neg \exists \sigma_j. \sigma_j \Rightarrow_\delta \sigma_j \land \sigma_j = \sigma_j) \right).
    \]
  - otherwise,
    \[
    \langle \vec{\sigma}, \vec{w}_Q \rangle \rightarrow_{\sigma, \vec{w}_Q} \langle \vec{\sigma}', \vec{w}_Q' \rangle \text{ if } \exists i. \sigma_i \Rightarrow_\delta \sigma_i \land \left( \forall j \neq i. \sigma_j \Rightarrow_\delta \sigma_j \lor (\neg \exists \sigma_j. \sigma_j \Rightarrow_\delta \sigma_j \land \sigma_j = \sigma_j) \right).
    \]

A run (or path) \( \rho \) of the system \( S \) is the following finite (or infinite) sequence of states.
\[
\rho : s_0 \rightarrow_{\delta_0} s_1 \rightarrow_{\delta_1} \cdots \rightarrow_{\delta_{n-1}} s_i \rightarrow_{\delta_i} s_i \rightarrow_{\delta_i} \cdots
\]
where \( \rightarrow_{\delta_i} \) between \( s_i \) and \( s_{i+1} \) is defined as
\[
s_i \rightarrow_{\delta_i} s_{i+1} \iff \exists \delta_i'. s_i \rightarrow_{\delta_i} s_i' \land s_i \rightarrow_{\delta_i} s_i' \rightarrow_{\delta_i} s_{i+1}.
\]

The initial state \( s_0 \) of a dynamically reconfigurable system is
\[
\langle (\sigma_{a_0}, \ldots, \sigma_{a_{|A|}}), (w_0, \ldots, w_{|Q|}) \rangle \text{ where each } \sigma_{a_i} \text{ is the initial state of DLHA } A_i \text{ and each } w_0 \text{ is empty; that is, } \forall j. w_0 = \epsilon.
\]

**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 \( S \) consisting of three DLHAs and one queue.

\[
A_1 = (L_1, V_1, Inv_1, Flow_1, Act_1, T_1, t_{01}, T_{d1}),
\]
\[
A_2 = (L_2, V_2, Inv_2, Flow_2, Act_2, T_2, t_{02}, T_{d2}),
\]
\[
A_3 = (L_3, V_3, Inv_3, Flow_3, Act_3, T_3, t_{03}, T_{d3}),
\]
\[
S = \{A_1, A_2, A_3, \{q\}\}
\]
where

\[ L_1 = \{ Run, Wait \} \]
\[ V_1 = \{ x \} \]
\[ Inv_1 = \{ Run \mapsto x \leq 10, Wait \mapsto true \} \]
\[ Flow_1 = \{ Run \mapsto \dot{x} = 1, Wait \mapsto \dot{x} = 0 \} \]
\[ Act_1 = \{ Dst_A?, start_t, q!A_i \} \]
\[ T_1 = \left\{ (Run, x \geq 10, q!A_i, \{ \}), Wait, Wait, Dst_A?, \{ x := 0 \}, Run \right\} \]
\[ t_{01} = \{ Run, start_t, \{ x := 0 \} \} \]
\[ T_{d1} = \{ \} \]
\[ L_2 = \{ Idle, Create \} \]
\[ V_2 = \{ y \} \]
\[ Inv_2 = \{ Idle \mapsto true, Create \mapsto y \leq 0 \} \]
\[ Flow_2 = \{ Idle \mapsto \dot{y} = 1, Create \mapsto \dot{y} = 0 \} \]
\[ Act_2 = \{ Crt_A!, start_t, q?A_i \} \]
\[ T_2 = \left\{ (Idle, true, q?A_i, \{ y := 0 \}, Create, Create, y \geq 0, Crt_A!, \{ \}, Idle) \right\} \]
\[ t_{02} = \{ Idle, start_t, \{ y := 0 \} \} \]
\[ T_{d2} = \{ \} \]
\[ L_3 = \{ Execute \} \]
This system runs as follows:
1) \( A_1 \) requires \( A_3 \) to be created from \( A_2 \) by enqueueing a message, and it waits for the message to return from \( A_1 \).
2) When \( A_2 \) receives the message, it creates \( A_3 \).
3) After \( A_3 \) finishes processing the job, it sends the message to \( A_1 \) and is destroyed.
4) This system infinitely repeats steps 1) to 3).

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

\[
\rho : \left\{ ((\text{Run}, x = 0), (\text{Idle}, y = 0), \bot), (\varepsilon) \right\}
\rightarrow_{q, A_1}^{y_0} \left\{ ((\text{Wait}, x = 10), (\text{Idle}, y = 0), \bot), (A_3) \right\}
\rightarrow_{q, A_2}^{y_1} \left\{ ((\text{Wait}, x = 10), (\text{Create}, y = 0, \bot), (\varepsilon) \right\}
\rightarrow_{\text{Crt}, A_3}^{y_2} \left\{ ((\text{Wait}, x = 10), (\text{Idle}, y = 0), (\text{Execute}, z = 0), (\varepsilon) \right\}
\rightarrow_{\text{Inv}, A_1}^{y_3} \left\{ ((\text{Run}, x = 0), (\text{Idle}, y = 0), \bot), (\varepsilon) \right\}
\rightarrow \cdots
\]

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 \( S = (A, Q) \) and a location \( l_1 \), \( S \) reaches \( l_1 \) if there exists a path such that

\[
s_0 \rightarrow_{a_0}^{y_0} \cdots \rightarrow_{a_{i-1}}^{y_{i-1}} s_i
\]

\( l_1 \), \( l \)

\[
\land \sigma_i = \left( \sigma_1, \cdots, \sigma_{|Q|}, \overrightarrow{w}_{Q} \right), \exists k. \text{loc}(\sigma_k) = l_1,
\]

where

\[
\text{loc}(\sigma) = \begin{cases} l & (\sigma = (l, v)) \\ \bot & (\text{undefined}) \end{cases}
\]

**Definition 9 (Reachability Problem).** Given a DRS \( S = (A, Q) \) and a location \( l_1 \), we output “yes” if \( S \) can reach \( l_1 \), 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 $V = \{x_1, \ldots, x_n\}$ of variables, a convex polyhedron $\zeta$ on $V$ is a $n$-dimensional real space. In particular, we define $true$ and $false$ as $true = \mathbb{R}^n$ and $false = \emptyset$.

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

$$\exists x_1, x_2 \geq 5 \land x_1 + x_2 = 1 \land true$$

$$= \exists x_1, x_2 \geq 5 \land x_1 + x_2 = 1$$

$$= x_2 \leq 1 \land x_2 \geq 4$$

$$= false.$$

### 4.2.2. Algorithm of Reachability Analysis

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

- $L$ is a finite set of locations.
- $\zeta$ is a convex polyhedron.
- $\bar{w}_Q$ 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 $S$ (ll.1-3).
2. Initialize a traversed set $Visit$ and a untraversed set $Wait$ of states by $\emptyset$ and $\{s_0\}$ (line 4).
3. While $Wait$ is not empty, repeat the following process (ll.5-16).
   (a) Take a state $(L, \zeta, \bar{w}_Q)$ from $Wait$ and remove the state from $Wait$ (ll.6-7).

**Input:** a system $S$ and a target location $l_t$

**Output:** “yes” or “no”

1. $L_0 \leftarrow \{l_0 \mid t_{l_0} = (l_{t_0}, a_{t_0}, \lambda_{l_0}), a_{t_0} \notin Crt.A_t\}$
2. $\lambda_0 \leftarrow \bigcup \{l_{t_0} \mid t_{l_0} = (l_{t_0}, a_{t_0}, \lambda_{l_0}), a_{t_0} \notin Crt.A_t\}$
3. $s_0 \leftarrow (L_0, 0|\lambda_{l_0}, (\varepsilon, \ldots, \varepsilon))$ /* Compute the initial state */
4. $Visit \leftarrow \emptyset$, $Wait \leftarrow \{s_0\}$ /* Initialize */
5. while $Wait \neq \emptyset$ do
6. $(L, \zeta, \bar{w}_Q) \leftarrow s \in Wait$
7. $Wait \leftarrow Wait \setminus \{(L, \zeta, \bar{w}_Q)\}$
8. if $l_t \in L$ then
9. return “yes”
10. end if
11. if $(L, \zeta, \bar{w}_Q) \notin Visit$ then
12. $Visit \leftarrow Visit \cup \{(L, \zeta, \bar{w}_Q)\}$
13. $S_{post} \leftarrow Succ((L, \zeta, \bar{w}_Q), S)$ /* 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.
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 ($\{L, \zeta, \bar{w}_Q\} \not\in \text{Visit}$) (line 11),
   i. add the state into \text{Visit} (line 12),
   ii. compute the set $S_{\text{post}}$ of successors by using the subroutine \text{Succ} (line 13), and
   iii. add all components of $S_{\text{post}}$ to \text{Wait} (line 14).

**Subroutine Succ** Figure 3 shows the subroutine \text{Succ} to compute the successors of a state. In this algorithm, we make the following assumptions.

$$S = (A, Q) = \left( \{A_1, \ldots, A_m\}, \{q_1, \ldots, q_k\} \right),$$

$$\text{Inv}_i = \bigcup_{k=1}^{m} \text{Inv}_{i_k},$$

where

$A_i = (L_i, V_i, \text{Inv}_{i_k}, \text{Flow}_{i_k}, \text{Act}_{i_k}, T_i, t_{oi}, T_{di})$ is a DLHA,

$t_{oi} = (l_{oi}, a_{oi}, \lambda_{oi}).$
Let the initial state of $S$ be \((\sigma_{01}, \ldots, \sigma_{0k}, \bar{w}_{0})\); \(s_0 = (L_0, \xi_0, \bar{w}_{0})\), where

\[
\text{zone}(\sigma) = \begin{cases} 
\nu & (\sigma = (l, \nu)) \\
\text{true} & (\sigma = \bot),
\end{cases}
\]

\[L_0 = \{ \text{loc}(\sigma_{0i}) | \sigma_{0i} \neq \bot, i \in \{1, \ldots, |A|\}\},\]

\[\xi_0 = \bigwedge_{i=0}^{|A|} \text{zone}(\sigma_{0i}).\]

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

\(T_{\text{succ}}(L, \xi)\) is a function that returns a convex polyhedron after performing a time transition on a given set \(L\) of locations and a convex polyhedron \(\xi\) (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

\[V_s = \bigcup_{i=1}^{\bar{m}} V_i = \{x_1, \ldots, x_n\} \quad \text{and} \quad \dot{V}_s = \{\dot{x}_1, \ldots, \dot{x}_n\}.
\]

\[T_{\text{succ}}(L, \xi) = \exists x_1, \ldots, x_n \in V_s, \exists \delta \in \mathbb{R} \exists \dot{x}_1, \ldots, \dot{x}_n \in \dot{V}_s.
\]
\[
\zeta \land \text{Flow}(L) \land \bigwedge_{x \in V} x' = x + \delta x \text{ and rename } x' \text{ as } x,
\]

where

\[
\text{Flow}_x = \bigcup_{k=1}^{4} \text{Flow}_{x_k},
\]

\[
\text{Flow} = \bigwedge_{x \in L} \text{Flow}(L).
\]

For a convex polyhedron \( \zeta \) and a set \( \lambda \) of update expressions, \( \zeta[\lambda] \) denotes the convex polyhedron updated by \( \lambda \) for \( \zeta \). Let the set of reset variables and set of shifted variables be \( V_r = \{ x \mid x := c \in \lambda \} = \{ x_{r_1}, \ldots, x_{r_m} \} \) and \( V_a = \{ x \mid x + c \in \lambda \} = \{ x_{a_1}, \ldots, x_{a_n} \} \). \( \zeta[\lambda] \) can be computed as

\[
\zeta[\lambda] = (\exists x_{r_1}, \ldots, x_{r_m} \in V_r, \zeta_a) \land x = m_r(x),
\]

where

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

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

\[
\zeta_a = \exists x_{a_1}, \ldots, x_{a_n} \in V_a, \zeta \land x' = x + m_a(x) \text{ and rename variables } x' \text{ as } x.
\]

Given a state \((L, \zeta, w_0)\) 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 \( S_{\text{post}} \) of post states is computed as follows (II.5-31):

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

\[
\zeta_\delta = \text{Tsucc}(L, \zeta) \land \text{Inv}_t(l_p).
\]

(b) If \( a \) is an internal action, \( S_{\text{post}} \) 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)

\[
\zeta' = (\zeta \land \phi)[\lambda] \land \text{Inv}_t(l'_p).
\]

iii. If \( a \) is an enqueue action \( q_{k!}w \) (II.11-15),

\[
S_{\text{post}} = \begin{cases} \{(L', \zeta', w_{q_{k!}}')\} & (\zeta' \neq \text{false}) \\
\emptyset & \text{(otherwise)} \end{cases},
\]

where

\[
w_{q_{k!}}' = (w_{k+1}, w_{k+2}, \ldots, w_{k!}, w_{k+1}, \ldots, w_{k+l}).
\]

iv. If \( a \) is a dequeue action \( q_{k?}w \) (II.16-20),
\[
S_{post} = \begin{cases}
\left\{ (L', \zeta', \bar{w}_\zeta^j) \right\} & (\zeta' \neq \text{false}, w_k = w \cdot w'_k, \forall j \neq k, \bar{w}_j = w'_j) \\
\emptyset & \text{(otherwise)}.
\end{cases}
\]

v. If \( a \) is another internal action (line 22),
\[
S_{post} = \begin{cases}
\left\{ (L', \zeta', \bar{w}_\zeta^j) \right\} & (\zeta' \neq \text{false}) \\
\emptyset & \text{(otherwise)}.
\end{cases}
\]

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

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

**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 \( (L, \zeta, \bar{w}_\zeta) \), a transition (or destruction-transition) \( t = (l, \phi, a_i!, \lambda, l') \), and a system \( S = (A, \mathcal{Q}) \), a set \( S_{post} \) of successors is computed as follows:

1. Initialize \( S_{post} \) as \( \emptyset \) (line 1).
2. Compute a convex polyhedron \( \zeta_\delta \) for the time transition (line 2).
\[
\zeta_\delta = \text{Tsucc}(L, \zeta) \cap \bigwedge_{\lambda \in \Lambda_\delta} T_{\text{inv}}(t).
\]
3. For each \( A_i \) in the system \( S \), compute the set \( T_{si} \) of transitions that are outgoing from the state by using an input action \( a_i \) (ll.3-5),
\[
T_{si} = \left\{ (l, \phi, a_i, ?, \lambda, l') \in T_i \mid l_i \in (L \setminus \{l\}) \right\}.
\]
4. Compute the set \( \Delta \) of combinations of \( T_{si} \) (line 6).
\[
\Delta = \prod \left\{ T_{si} \mid T_{si} \neq \emptyset, i \in \{1, \ldots, |A|\} \right\}.
\]
5. For each combination \( T = (t_1, \ldots, t_n) \in \Delta \), the successor \( (L'_1, \zeta'_1, \bar{w}_{\zeta'_1}) \) is computed as follows (ll.7-29):
   
   (a) Compute the set \( T_{\text{sync}} \) of transitions (line 9).
   \[
   T_{\text{sync}} = \max_{|k|} \Delta' \subseteq \{t_1, \ldots, t_n\}
   \]
   s.t. \( \zeta_\delta \wedge \phi \wedge \bigwedge \{ \phi_i \mid (l_1, \phi, a_i, ?, \lambda, l_2) \in \Delta' \} \neq \text{false}.
   
   (b) Compute the set \( L'_i \) of locations (ll.9-14, line 21).
   \[
   L'_i = (L \setminus L_{\text{pre}}) \cup L_{\text{post}}.
   \]
where
\[
L_{\text{sync}} = \left\{ l_i \mid (l_i, \phi, a_i, ?, \lambda, l_2) \in T_{\text{sync}} \right\},
\]
\[
L'_{\text{sync}} = \left\{ l'_i \mid (l'_i, \phi, a_i, ?, \lambda, l_2) \in T_{\text{sync}} \right\},
\]
\[
L_{\text{pre}} = \{ t \} \cup L_{\text{sync}}.
\]
\[
L_{\text{post}} = \begin{cases}
\{ l'_i, l_{j0} \} \cup L'_{\text{sync}} & (a_i = \text{Crt}_A, L \cap L_t = \emptyset) \\
L'_{\text{sync}} & (a_i = \text{Dst}_A) \\
\{ l'_i \} \cup L'_{\text{sync}} & \text{(otherwise)}.
\end{cases}
\]
(c) Compute the update expression $\lambda_{\text{sync}}$ (ll.9-14).

$$\lambda_{\text{sync}} = \begin{cases} 
\lambda \cup \lambda_{0} \cup \lambda_{\text{in}} & (a_i = \text{Crt}_A, L \cap L_j = \emptyset) \\
\lambda_{\text{in}} & (a_i = \text{Dst}_A) \\
\lambda \cup \lambda_{\text{in}} & (\text{otherwise}),
\end{cases}$$

where

$$\lambda_{\text{in}} = \bigcup \{ \lambda_x \mid (l_i, \phi_x, a_i, \lambda_x, l_2) \in T_{\text{sync}} \}.$$ 

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

$$\phi_{\text{sync}} = \phi \land \bigwedge \{ \phi_x \mid (l_i, \phi_x, a_i, \lambda_x, l_2) \in T_{\text{sync}} \}.$$ 

(e) Compute the convex polyhedron $\zeta'$ (ll.22-24).

$$\zeta' = \begin{cases} 
\exists x, j, \cdots, x, \exists \phi \mid V_j, \zeta' & (a_i = \text{Dst}_A) \\
\zeta' & (\text{otherwise}),
\end{cases}$$

where

$$\zeta' = (\zeta \land \phi_{\text{sync}})[\lambda_{\text{sync}}] \land \bigwedge_{l'_p \in L'} \\text{Inv}_{l'_p} (l'_p).$$

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

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

**Lemma 1.** If the algorithm terminates and returns “$l_i$ is not reachable”, the system $S$ has the safety property.

**Lemma 2.** If this algorithm terminates and returns “$l_i$ is reachable”, the system $S$ 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 \textit{asap} (that means “as soon as possible”). For a transition labeled \textit{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.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{model_checker.png}
\caption{Model checker for DRSs.}
\end{figure}

\begin{verbatim}
target: Execute

DLHA:
\begin{verbatim}
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
}
\end{verbatim}
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{example_input_file.png}
\caption{Example input file: description for checking the reachability of the system in Figure 1.}
\end{figure}
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.](image)

**Table 1. Parameters of tasks.**

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Deadline</th>
<th>Priority</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>70 ms</td>
<td>70 ms</td>
<td>high</td>
<td>20 ms, co-task a0, 10 ms, co-task b0</td>
</tr>
<tr>
<td>B</td>
<td>200 ms</td>
<td>200 ms</td>
<td>low</td>
<td>co-task a1, 97 ms</td>
</tr>
</tbody>
</table>
Table 2. Parameters of co-tasks.

<table>
<thead>
<tr>
<th>co-task</th>
<th>Processing time</th>
<th>Deadline</th>
<th>Tiles</th>
<th>Rate of Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0, a1</td>
<td>10 ms</td>
<td>15 ms</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>b0</td>
<td>5 ms</td>
<td>10 ms</td>
<td>6</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Figure 8. Components of the system.

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_a0! and moves to RunA from WaitA, The Sender receives Act_Create_a0? and enqueues cotask_a0 in q with q! cotask_a0.

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.

---

**Figure 13. Task: TaskB.**
Figure 14. CPU Scheduler: Scheduler.

Figure 15. Message sender to DRP: Sender.

Figure 16. DRP_Dispatcher.

Figure 17. Co-task: cotask_a0.
**Figure 18. Co-task: cotask_a1.**

- $C_{rt\_cotask\_a1}$?
  - \( e_{a1} := 0, r_{a1} := 0 \)
- $Run\_a1$
  - \( e_{a1} \leq 10 \)
  - \( r_{a1} = 1 \)
- $e_{a1} \geq 10$, $Dist\_cotask\_a1$

**Figure 19. Co-task: cotask_b0.**

- $C_{rt\_cotask\_b0}$?
  - \( e_{b0} := 0, r_{b0} := 0 \)
- $Run\_b0$
  - \( e_{b0} \leq 5 \)
  - \( r_{b0} = 1 \)
- $e_{b0} \geq 5$, $Dist\_cotask\_b0$

**Figure 20. Frequency_Manager.**
Table 3. Modified parameters of tasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Deadline</th>
<th>Priority</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>90 ms</td>
<td>80 ms</td>
<td>high</td>
<td>20 ms, co-task b0,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20 ms, co-task a0</td>
</tr>
<tr>
<td>B</td>
<td>200 ms</td>
<td>150 ms</td>
<td>low</td>
<td>co-task a1, 70 ms</td>
</tr>
</tbody>
</table>

Table 4. Modified parameters of co-tasks.

<table>
<thead>
<tr>
<th>co-task</th>
<th>Processing time</th>
<th>Deadline</th>
<th>Tiles</th>
<th>Rate of Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0, a1</td>
<td>5 ms</td>
<td>10 ms</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>b0</td>
<td>10 ms</td>
<td>20 ms</td>
<td>5</td>
<td>1/3</td>
</tr>
</tbody>
</table>

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, $E_A - e_A > D_A - r_A \iff 30 - e_A > 70 - r_A \iff r_A - e_A > 40$ 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 \( create_{a0} \) is received from task A, the monitor automaton starts counting time for co-task \( a0 \). If the waiting time exceeds the deadline of task A before it receives the message \( Crt_{cotask_{a0}} \), the monitor moves to error location. **Figure 22** shows The monitor automaton for the case of **Table 1** for co-task \( a0 \). Monitor automata for co-tasks \( a1 \) and \( b0 \) can be similarly described.

**Figure 21.** Monitor automaton for checking schedulability.

**Figure 22.** Monitor automaton for checking creation of co-task \( a0 \).
5.3.3. Destruction of Co-Tasks
Each co-task must be destroyed before the waiting time reaches its deadline. For the co-task a0, when the message \( \text{Crt\_cotask\_a0} \) is received from the dispatcher DRP\_Dispatcher, the monitor automaton checks the message \( \text{Dst\_cotask\_a0} \). 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 a0 and co-task b0 are running on the DRP, Frequency\_Manager must be at location \( L\_Freq\_b \). Figure 24 shows the monitor automaton for the case of Table 2.
Table 5. Experimental results.

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

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


Submit or recommend next manuscript to SCIRP and we will provide best service for you:

Accepting pre-submission inquiries through Email, Facebook, LinkedIn, Twitter, etc.
A wide selection of journals (inclusive of 9 subjects, more than 200 journals)
Providing 24-hour high-quality service
User-friendly online submission system
Fair and swift peer-review system
Efficient typesetting and proofreading procedure
Display of the result of downloads and visits, as well as the number of cited articles
Maximum dissemination of your research work

Submit your manuscript at: [http://papersubmission.scirp.org/](http://papersubmission.scirp.org/)
Or contact jsea@scirp.org