Dynamic Linear Hybrid Automata and Their Applications to Formal Verification of Dynamic Reconfigurable Embedded Systems

Abstract for Dissertation

Dynamic Linear Hybrid Automata and Their Applications to Formal Verification of Dynamic Reconfigurable Embedded Systems

動的線形ハイブリッドオートマタと動的再構成可能組込みシステムの形式検証への応用

Ryo YANASE

(Student Number: 1323112012)

Graduate School of Natural Science and Technology

Division of Electrical Engineering and Computer Science

Kanazawa University

Chief supervisor: **Prof. Satoshi YAMANE**

January 2017

Abstract

Networking systems and embedded systems are able to change their configuration, components and modules at run-time. Such a system is called dynamically reconfigurable system. For guaranteeing safety of the system, model checking is one of the effective methods. This paper presents a dynamic linear hybrid automaton (DLHA) as a specification language for designing dynamically reconfigurable systems. As a practical experiment, we describe an embedded cooperative system consisting of CPU and DRP by DLHAs and verify several properties for the system with a model checker that performs the reachability analysis by using monitor automata.

1 Introduction

Dynamically reconfigurable systems are being used in a number of areas [1, 2, 3]. The major methods of checking system safety include simulation and testing; however, it is often difficult for them to ensure safety precisely, since these methods don't check all states. In such cases, model checking 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 Features of dynamically reconfigurable systems consisting of CPU and DRP

The target of our research is an embedded system in which a CPU and dynamically reconfigurable hardware, e.g., DRP or D-FPGA [4] operate cooperatively. The dynamically reconfigurable processor (DRP) is a coarse-grained programmable processor developed by NEC [3] and it manages both the power conservation and miniaturization. The DRP is used to accelerate the computations of a general purpose CPU with through cooperating operations, and it has the following features:

- Dynamically creation/destruction of the function: 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.

For the experiments, we specified a dynamically reconfigurable embedded system consisting of a CPU and DRP, and verified the 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

1.2.1 Specification

We developed a new specification language (DLHA) based on a linear hybrid automaton [5] with both creation/destruction events and unbounded FIFO queues. DLHA is different from existing research in the following points:

- V. Varshavsky and others proposed the GALA (Globally Asynchronous - Locally Arbitrary) modeling approach including timed guards [6]. 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[7]. Since linear hybrid automata cannot describe changes to the configuration and asynchronous communications by using unbounded FIFO queues, the system has been specified as a static system.
- P. C. Attie and N. A. Lynch specified systems whose components are dynamically created/destroyed by using I/O automata [8]. I/O automata cannot describe changes in variables, for example, changes in clock and operating frequency.

- H. Yamada and others proposed hierarchical linear hybrid automata for specifying dynamically reconfigurable systems [9]. They introduced concepts such as class, object, etc., to the specification language. However, as the scale of a 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 [10]. 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 symbolic reachability analysis for FIFO-channel systems [11, 12]. However, since their analysis don't provide for continuous changes in variables, in languages cannot be used for designing hybrid systems.

1.2.2 Verification Method

The originality of our work on the verification method twofold:

- Our method targets systems that dynamically change their configurations, which is something the existing work, such as HyTech, 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.

2 Dynamic Linear Hybrid Automaton

2.1 Syntax

A dynamic linear hybrid automaton (DLHA) is an extended linear hybrid automaton and represented as a 8-tuple $(L, V, Inv, Flow, Act, T, t_0, T_d)$, where

- L is a finite set of nodes called locations.
- V is a finite set of variables.
- $Inv : L \to \Phi(V)$ is a function that assigns an invariant to each location, where $\Phi(V)$ is a set of all constraints over V.
- $Flow : L \to F(V)$ is a function that assigns a flow condition to each location, where $F(V)$ is a set of all flow conditions over V.
- $Act = Act_{in} \cup Act_{out} \cup Act_{\tau}$ 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 broadcasting the message m to each DLHA.
  - Act_{τ} is a finite set of *internal actions* that denote other events.

  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 the DLHA $\mathcal{A}'$. $Crt\_\mathcal{A}'? \in Act_{out}$ 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 queue q.

- $T \subseteq L \times \Phi(V) \times Act \times 2^{UPD(V)} \times L$ is a finite set of edges called *transitions*. Here, a constraint $\phi \in \Phi(V)$ is called a *guard condition*, and $\lambda \in 2^{UPD(V)}$ are called *update expressions*. Each update expression has the form $x := c$ or $x := x + c$, where $x \in V$ and $c \in \mathbb{Q}$.
- $t_0 \in L \times (Act_{in} \cup Act_\tau) \times 2^{UPD(V)}$ is an *initial transition*.
- $T_d \subseteq L \times \Phi(V) \times Act_{out}$ is a finite set of *destruction-transitions*.

2.2 Operational Semantics

A state σ of a DLHA $(L, V, Inv, Flow, A, T, t_0, T_d)$ is defined as $\bot \mid (l, \nu)$, where $l \in L$ is a location, $\nu : V \to \mathbb{R}$ is an assignment called *evaluation* of variables, and $\bot$ denotes an *undefined value*.

The semantics $\mathcal{M}$ of the DLHA is defined as $(\Sigma, \Rightarrow, \sigma_0)$, where Σ is a set of states, $\Rightarrow$ is a set of *time transitions* and *discrete transitions* and σ_0 is the initial state.

2.2.1 Time transition

For arbitrary $\delta \in \mathbb{R}_{\geq 0}$,

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

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

2.2.2 Discrete transition

For an evaluation ν and update expressions $\lambda \in 2^{UPD(V)}$, $\nu[\lambda]$ denotes an evaluation updated by λ.

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

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

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

3 Dynamically Reconfigurable Systems

To describe an asynchronous communication among DLHAs in a dynamically reconfigurable system, 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.

A dynamically reconfigurable system $\mathcal{S} = (A, \mathcal{Q})$ consists of a finite set $A = \{\mathcal{A}_1, \ldots, \mathcal{A}_{|A|}\}$ of DLHAs and a finite set $\mathcal{Q} = \{q_1, \ldots, q_{|\mathcal{Q}|}\}$ of queues.

A state s of the dynamically reconfigurable system is a tuple $\langle \vec{\sigma}, \vec{w}_\mathcal{Q} \rangle$ where $\vec{\sigma}$ is a vector of states of DLHAs and $\vec{w}_\mathcal{Q}$ is a vector of contents of queues.

3.0.1 Time Transition

For an arbitrary $\delta \in \mathbb{R}_{\geq 0}$, the time transition is defined as

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

3.0.2 Discrete Transition

Let $\vec{\sigma}, \vec{\sigma}', \vec{w}_\mathcal{Q}$ and $\vec{w}'_\mathcal{Q}$ be $\vec{\sigma} = (\sigma_1, \ldots, \sigma_{|A|})$, $\vec{\sigma}' = (\sigma'_1, \ldots, \sigma'_{|A|})$, $\vec{w}_\mathcal{Q} = (w_1, \ldots, w_{|\mathcal{Q}|})$ and $\vec{w}'_\mathcal{Q} = (w'_1, \ldots, w'_{|\mathcal{Q}|})$.

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

  $$\text{iff } \exists i. \sigma_i \Rightarrow_{a!} \sigma'_i \wedge \forall j \neq i. \sigma_j \Rightarrow_{a?} \sigma_j \vee ((\neg \exists \sigma'_j. \sigma_j \Rightarrow_{a?} \sigma'_j) \wedge \sigma_j = \sigma'_j).$$

  An output action is broadcasted to all DLHAs, and a DLHA receiving the action moves by synchronization if the the guard condition holds in the state.
- For an internal action a_τ,
  - in the case of $a_\tau = q_k!w$, $\langle \vec{\sigma}, \vec{w}_\mathcal{Q} \rangle \to_{q_k!w} \langle \vec{\sigma}', \vec{w}'_\mathcal{Q} \rangle$,

    $$\text{iff } \exists i. \sigma_i \Rightarrow_{q_k!w} \sigma'_i \wedge \forall j \neq i. \sigma_j = \sigma'_j \wedge w'_k = w_k w \wedge \forall l \neq k. w_k = w'_k,$$

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

$$\text{iff } \exists i.\sigma_i \Rightarrow_{q_k?w} \sigma'_i \wedge \forall j \neq i.\sigma_j = \sigma'_j \wedge w_k = ww'_k \wedge \forall l \neq k.w_l = w'_l,$$

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

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

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

where $\rightarrow^{\delta_i}_{a_i}$ between s_i and s_{i+1} is defined as follows:

$$s_i \quad \rightarrow^{\delta_i}_{a_i} s_{i+1} \iff \exists s'_i.s_i \rightarrow_{\delta_i} s'_i \wedge s'_i \rightarrow_{a_i} s_{i+1}.$$

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

4 Reachability Analysis

4.1 Reachability Problem

We define reachability and the reachability problem for a dynamically reconfigurable system as follows:

Definition 1 (Reachability) *For a dynamically reconfigurable system $\mathcal{S} = (A, \mathcal{Q})$ and a location l_t, $\mathcal{S}$ reaches l_t if there exists a path $s_0 \rightarrow^{\delta_0}_{a_0} \cdots \rightarrow^{\delta_{t-1}}_{a_{t-1}} s_t$ such that s_t has a DLHA-state which contains the location l_t.*

Definition 2 (Reachability Problem) *Given a dynamically reconfigurable system $\mathcal{S} = (A, \mathcal{Q})$ and a location l_t, we output "yes" if $\mathcal{S}$ can reach l_t, and "no" otherwise.*

4.2 Algorithm of Reachability Analysis

Fig. 1 show the algorithm of the reachability analysis. Our method introduces *convex polyhedra* for the reachability analysis in accordance with [17]. In this algorithm, we define a state s in the reachability analysis as $(L, \zeta, \vec{w}_{\mathcal{Q}})$, where L is a finite set of locations, ζ is a convex polyhedron, and $\vec{w}_{\mathcal{Q}}$ is a vector of contents of queues. Fig.1 is an overview of the reachability analysis, and this algorithm is performed by using the extended method of [13] with a set $\mathcal{Q}$ of queues. The analysis is performed as follows:

1. Compute the 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 $(L, \zeta, \vec{w_{\mathcal{Q}}})$ from Wait and remove the state from Wait (ll.6–7).
   (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 ($(L, \zeta, \vec{w_{\mathcal{Q}}}) \notin$ Visit) (line 11),
      i. add the state to 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).

The subroutine Succ computes successors of a state. Successors for a state s together with a transition that has an output action are computed by the following procedures:

1. Initialize S_{post} by $\varnothing$.
2. Compute a convex polyhedron ζ_δ for time transition.
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 the input action $a_l?$.
4. Compute a set Δ of combinations of T_{si}.
5. For each combination $T = (t_1, \ldots, t_n) \in \Delta$, the successor $s' = (L'_T, \zeta'_T, \vec{w}_{\mathcal{Q}})$ is computed and $S_{post} := S_{post} \cup \{s'\}$.

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

Lemma 1 *If this algorithm terminates and returns "l_t is not reachable", the system $\mathcal{S}$ holds 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 dynamically reconfigurable systems is undecidable, and this algorithm might not terminate [13].

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.

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: $(L, \zeta, \vec{w}_\mathcal{Q}) \leftarrow s \in$ Wait
7: Wait $\leftarrow$ Wait $\setminus \{(L, \zeta, \vec{w}_\mathcal{Q})\}$
8: **if** $l_\mathrm{t} \in L$ **then**
9: **return** "yes"
10: **end if**
11: **if** $(L, \zeta, \vec{w}_\mathcal{Q}) \notin$ Visit **then**
12: Visit $\leftarrow$ Visit $\cup \{(L, \zeta, \vec{w}_\mathcal{Q})\}$
13: $S_{post} \leftarrow$ Succ$((L, \zeta, \vec{w}_\mathcal{Q}), \mathcal{S})$ /* Compute the set of post-states */
14: Wait $\leftarrow$ Wait $\cup S_{post}$
15: **end if**
16: **end while**
17: **return** "no"

Figure 1: Reachability Analysis

5 Practical Experiment

5.1 Model Checker

We implemented a model checker of dynamically reconfigurable systems consisting of DLHAs in Java (about 1,600 lines of code) by using the LAS, PPL, and QDD external libraries [10, 14, 15, 16]. 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)". The monitor automaton had a special location (we call it the error location), and checked the system without changing the system's behavior [17]. 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.

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 [7]. 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.

Fig. 2 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 1-2.

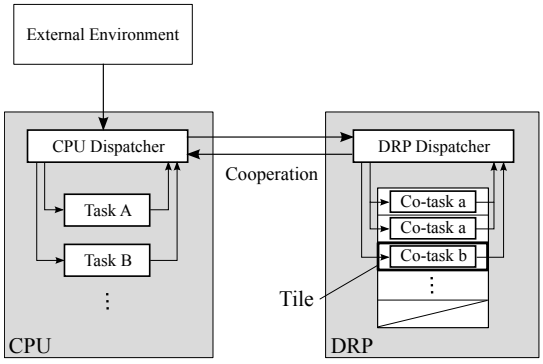

Figure 2: Overview of the CPU-DRP embedded system

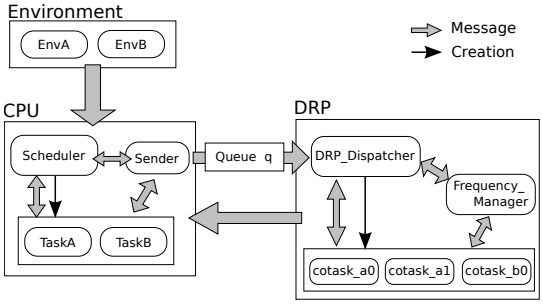

Figure 3: Components of the system

The system, whose components are illustrated in Fig.3, consists of 11 DLHAs and 1 queue. The

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
$a0, a1$	10 ms	15 ms	2	1
$b0$	5 ms	10 ms	6	1/2

external environment consists of EnvA and EnvB that periodically create TaskA and TaskB. That is, EnvA creates TaskA every 70 milliseconds, and EnvB creates TaskB with every 200 milliseconds. The Scheduler performs scheduling in accordance with the priority and actions for creation and destruction of DLHAs.

TaskA and TaskB send a message to The Sender if they need a co-task. The Sender enqueues the message to create a co-task to q when it receives a message from tasks.

The DRP_Dispatcher dequeues a message and creates cotask_a0, cotask_a1, and cotask_b0 if there are enough free tiles. The Frequency_Manager 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. Parameters of the tasks are the same as those in Table 1.

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
$a0, a1$	5 ms	10 ms	4	1
$b0$	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. The verification experiment was performed on a machine with an Intel (R) Core (TM) i7-3770 (3.40GHz) CPU and 16GB RAM running Gentoo Linux (3.10.25-gentoo).

Verification properties are below:

- 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; 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 $E_A - e_A > D_A - r_A$ (Fig. 4).

- 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 $a0$. If the waiting time exceeds the deadline of task A before it receives the message Crt_cotask_a0, the monitor moves to error location.

- 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 *Crt_cotask_a0* is received from the dispatcher DRP_Dispatcher, the monitor automaton checks the message *Dst_cotask_a0*.

- 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*.

- 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.

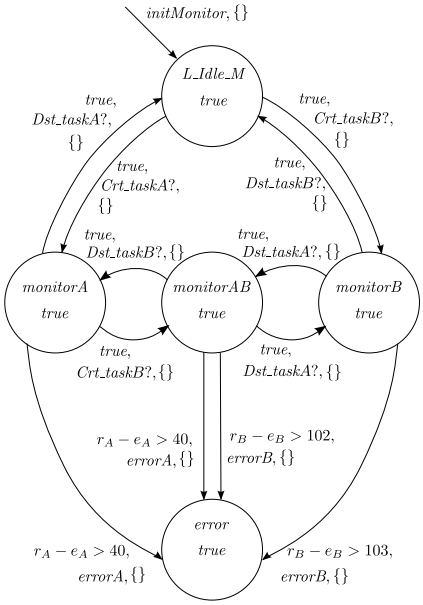

Figure 4: Monitor automaton checking schedulability

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.

6 Conclusion and future work

In this paper, 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) [18, 19].

References

[1] P. Garcia, K. Compton, M. Schulte, E. Blem and W. Fu. An Overview of Reconfigurable Hardware in Embedded Systems. *EURASIP J. Embedded Syst.*, 2006(1):1–19, 2002.

[2] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick and T. Brooks. Internet Worm and Virus Protection in Dynamically Reconfigurable Hardware. *In Military and Aerospace Programmable Logic Device (MAPLD)*, E10, 2003.

[3] M. Motomura, T. Fujii, K. Furuta, K. Anjo, Y. Yabe, K. Togawa, J. Yamada, Y. Izawa and R. Sasaki. New Generation Microprocessor Architecture (2):Dynamically Reconfigurable Processor (DRP). *IPSJ Magazine*, 46(11):1259–1265, 2005.

[4] H. Amano, Y. Adachi, S. Tsutsumi and K. Ishikawa. A context dependent clock control mechanism for dynamically reconfigurable processors. *Technical Report of IEICE*, 104(589):13–16, 2005.

[5] R. Alur, C. Courcoubetis, T. A. Henzinger and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Lecture Notes in Computer Science*, 736:209–229, 1993.

[6] V. Varshavsky and V. Marakhovsky. GALA (Globally Asynchronous – Locally Arbitrary) Design. *Lecture Notes in Computer Science*, 2549:61–107, 2002.

[7] S. Minami, S. Takinai, S. Sekoguchi, Y. Nakai and S. Yamane. Modeling, Specification and Model checking of dynamically reconfigurable processors. *Computer Software*, 28(1):190–216, 2011.

[8] P. C. Attie and N. A. Lynch. Dynamic input/output automata, a formal model for dynamic systems. *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing (PODC '01)*, 2154:314–316, 2001.

[9] H. Yamada, Y. Nakai and S. Yamane. Proposal of Specification Language and Verification Experiment for Dynamically Reconfigurable System. *Journal of Information Processing Society of Japan, Programming*, 6(3):1–19, 2013.

[10] B. Boigelot and P. Godefroid. Symbolic Verification of Communication Protocols with Infinite StateSpaces using QDDs. *Form. Methods Syst. Des.*, 14(3):237–255, 1999.

Table 5: Experimental results

Model	Property	Satisfiability	Memory [MB]	Time [sec]	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

[11] A. Bouajjani, J. Esparza and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. *Lecture Notes in Computer Science*, 1243:135–150, 1997.

[12] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. *Lecture Notes in Computer Science*,1256:560–570, 1997.

[13] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[14] Y. Ono and S. Yamane. Computation of quantifier elimination of linear inequlities of first order predicate logic. *IEICE Technical Report. COMP, Computation*, 111(20): 55–59, 2011.

[15] R. Bagnara, P. M. Hill and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1–2): 3–21, 2008.

[16] B. Boigelot, P. Godefroid, B. Willems and P. Wolper. The Power of QDDs (Extended Abstract). *SAS*, 172-186, 1997.

[17] T. A. Henzinger, P. Ho and H. Wong-toi. HyTech : A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, 1: 460–463, 1997.

[18] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. Counterexample-Guided Abstraction Refinement. *Proceedings of the 12th International Conference on Computer Aided Verification*, 1855:154–169, 2000.

[19] R. Nieuwenhuis, A. Oliveras and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. *In LPAR' 04, LNAI 3452*, 36–50, 2005.

平成２９年１月３１日

学位論文審査報告書（甲）

１．学位論文題目（外国語の場合は和訳を付けること。）

Dynamic Linear Hybrid Automata and Their Applications to Formal Verification of Dynamic Reconfigurable Embedded Systems

（動的線形ハイブリッドオートマタとその動的再構成可能組込みシステムの形式的検証への適用）

２．論文提出者　(1) 所　　属　電子情報科学専攻

(2) 氏（ふり）　名（がな）　柳瀬龍（やなせりょう）

３．審査結果の要旨（600～650 字）

平成 29 年 1 月 31 日に第 1 回学位論文審査委員会を開催、同日に口頭発表を実施し、その後に第 2 回学位論文審査委員会を実施した。慎重審議の結果、以下の通り判定した。なお、口頭発表における質疑を最終試験に替えるものとした。

汎用 CPU と動的再構成プロセッサが協調動作して、回路の機能及び動作周波数が動的に変化する動的再構成可能組込みシステムの仕様記述言語と形式的検証手法を開発し、その検証システムを実装して、提案手法の有効性を実証した。具体的には以下の(1)～(3)のとおりである：(1)仕様記述言語では、オートマトン理論を用いて、並列性と微分方程式が表現可能なハイブリッドオートマタを拡張して、動的にプロセスが生成消滅する動的ハイブリッドオートマタの形式的な構文と操作的意味を開発した。(2)形式的検証手法では、数理論理学を用いて、第一階述語論理により実数体上のベクトル空間の凸集合を記述して、実数体上のタルスキーの量化記号消去などの定理証明理論に基づいた、動的ハイブリッドオートマタの記号モデル検査手続きを開発した。(3)Java 言語により検証システムを開発して、オートマトンレベルの事例により、提案手法の有効性を実証した。

以上は、動的再構成可能組込みシステムの形式手法に関する、新規性の高い理論と実証に関する研究であり、博士（工学）に値する。

４．審査結果　(1) 判　　定（いずれかに○印）　○合　格　・　不合格

(2) 授与学位　　博　士（工学）