Formal verification of dynamically reconfigurable systems

Formal Verification of Dynamically Reconfigurable Systems

Ryo Yanase *, Tatsunori Sakai *, Makoto Sakai * and Satoshi Yamane †
* Graduate School of Natural Science and Technology
Kanazawa University, Kakuma-machi 920–1192
Email: ryanase@csl.ec.t.kanazawa-u.ac.jp
† Institute of Science and Engineering
Kanazawa University, Kakuma-machi 920–1192
Email: syamane@is.t.kanazawa-u.ac.jp

***Abstract*—A dynamically reconfigurable system can perform complicated operations with dynamically changing the configuration. For ensuring the safety of the system, a model checking is one of the efficient formal approach. In our work, we define the specification language of a dynamically reconfigurable system and propose the model checking algorithm of verifying safety properties.**

I. Introduction

If a system changes its configuration during the operation, we call it *dynamically reconfigurable system*. The feature of a dynamically reconfigurable system is appeared in many real-time systems (e.g., embedded systems, networks, etc.). Since the operation of a such system is generally complex, it is difficult to guarantee that the system is safe. Model checking is a formal method for verification, and it is one of the effective approach. In our work, we define the specification language *dynamic linear hybrid automaton (DLHA)* of dynamically reconfigurable systems and propose an approach to the model checking of safety properties.

II. Related Works

Linear hybrid automaton provides continuous and discrete operations but cannot describe the dynamic change of the configuration[1]. Therefore, the system is modeled as a static system[2]. Dynamic Input/Output automaton describes asynchronous concurrent systems with FIfO channels and also provides the dynamic change as creation and destruction of automata[3]. However, this language doesn't provide continuous transition and cannot describe a real-time system. Hierarchical linear hybrid automaton is based on concepts of object-orientation[4]. For this language, given a large scale system, the specification and the verification method tend to be complex. H. Nakano and others have developed a dynamically configurable processor LSI that the circuit configuration is changed by a single clock[5]. In our work, we assume that a dynamically reconfigurable system can change its configuration by a single clock.

III. Specification Language

We propose a specification language *Dynamic Linear Hybrid Automaton* (DLHA). A DLHA is a hybrid automaton extended by creation/destruction of automata and queue operations.

A. Dynamic Linear Hybrid Automaton

A *dynamic linear hybrid automaton*(DLHA) is a linear hybrid automaton extended with special actions that are labels representing the types of transitions.

1) Syntax of a DLHA: A DLHA is a tuple consisting of the following components:

- A finite set L of locations
- A finite set V of variables
- A function *Inv* that assigns a constraint to each location: A constraint ϕ on V is defined by

  $$\phi ::= \mathit{true} \mid x \sim e \mid x - y \sim e \mid \phi_1 \wedge \phi_2$$

  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 a set of all constraints on V. For a location $l \in L$, a constraint $\mathit{Inv}(l)$ is called *invariant* of l.
- A function *Flow* that assigns a *flow condition* to each location: Let $V = \{x_1, \ldots, x_n\}$ be a finite set of (real-valued) variables. A *flow condition* f on V is defined by

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

  where $d_1, \ldots, d_n \in \mathbb{Q}$. We can also write a set of all flow conditions in $F(V)$.
- A finite set *Act* of actions: An action is either one among *input action*, *output action* and *internal action*. An input action that has the form $m?$ represents receiving the message m. An output action that has the form $m!$ represents sending (broad-casting) the message m. An internal action a_τ represents an asynchronous transition between two locations. In particular, the special actions are below:
  - *Creation actions*: *Crt_*$\mathcal{A}_i!$ and *Crt_*$\mathcal{A}_i?$ denote the creation of $\mathcal{A}_i$.
  - *Destruction actions*: *Dst_*$\mathcal{A}_i!$ and *Dst_*$\mathcal{A}_i?$ denote the destruction of $\mathcal{A}_i$.
  - *Enqueue actions*: $q_i!m$ denotes enqueueing the message m into the queue q_i.
  - *Dequeue actions*: $q_i?m$ denotes dequeueing the message m from the queue q_i.

- A finite set $T \subseteq L \times \Phi(V) \times Act \times 2^{UPD(V)} \times L$ of transitions: A constraint of transition $g \in \Phi(V)$ is called *guard condition*.
  $UPD(V)$ is a set of *update expressions*. An update expression λ is defined by

  $$\lambda ::= x := c \mid x := x + c$$

  where $x \in V$ and $c \in \mathbb{Q}$.
- An initial transition $t_0 \in L \times (Act_{in} \cup Act_{\tau}) \times 2^{UPD(V)}$
- A finite set $T_{end} \subseteq L \times \Phi(V) \times Act_{out}$ of *destruction transitions*

2) Semantics of a DLHA: A state σ of a DLHA is defined as (l, ν) or $\perp$, where l is a location, ν is an *evaluation* of variables and $\perp$ is an undefined state (that is, the DLHA is not created). An evaluation is an assignment of variables to real numbers.

The operational semantics of a DLHA is defined by the following rules:

- Time transition:
  - For any $d \in \mathbb{R}_{\geq 0}$, $\perp \Rightarrow_d \perp$.
  - For $d \in \mathbb{R}_{\geq 0}$ and a state (l, ν),

    $$(l, \nu) \Rightarrow_d (l, \nu + d) \text{ if } \nu + d \in Inv(l)$$

    where $\nu + d$ is a shorthand for a function defined by $\nu(x) + Flow(x) \cdot d$.
- Discrete transition:
  - For a transition $(l, \phi, a, \lambda, l') \in T$,

    $$(l, \nu) \Rightarrow_a (l', \nu[\lambda]) \text{ if } \nu \in \phi, \nu[\lambda] \in Inv.$$

  - For a destruction-transition $(l, \phi, a) \in T_{end}$,

    $$(l, \nu) \Rightarrow_a \perp \text{ if } \nu \in \phi.$$

  - For the initial transition $t_0 = (l_0, a_0, \lambda_0)$,

    $$\perp \Rightarrow_{a_0} (l_0, \vec{0}[\lambda])$$

    where $\vec{0}$ is an evaluation that assigns 0 to each variables.

B. Dynamically Reconfigurable System

A dynamically reconfigurable system consists of a set of DLHAs and a set of queues. Formally, a system $\mathcal{S}$ is defined by a tuple $(A, \mathcal{Q})$, where A is a finite set of DLHAs and $\mathcal{Q}$ is a finite set of queues (unbounded FIfO buffers). A state of the system is a pair $\langle \vec{\sigma}, \vec{w_{\mathcal{Q}}} \rangle = \langle (\sigma_1, \ldots, \sigma_{|A|}), (w_1, \ldots, w_{|\mathcal{Q}|}) \rangle$ of a vector of DLHA-states and a vector of queue-contents.

1) Time Transition: For an arbitrary $\delta \in \mathbb{R}_{\geq 0}$, the time transition is defined by the following rule:

$$\langle \vec{\sigma}, \vec{w_{\mathcal{Q}}} \rangle \rightarrow_{\delta} (\vec{\sigma'}, \vec{w}_{\mathcal{Q}}) \iff \forall i.[\sigma_i \Rightarrow_{\delta} \sigma_i].$$

2) Discrete Transition: Let $\vec{\sigma}, \vec{\sigma'}, \vec{w}_{\mathcal{Q}}$ and $\vec{w'}_{\mathcal{Q}}$ be

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

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

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

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

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

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

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

  - Otherwise, $\langle \vec{\sigma}, \vec{w}_{\mathcal{Q}} \rangle \rightarrow_{a_{\tau}} \langle \vec{\sigma'}, \vec{w}_{\mathcal{Q}} \rangle$

    $$\text{if } \exists i.[\sigma_i \Rightarrow_{a_{\tau}} \sigma'_i \wedge \forall j \neq i.[\sigma_j = \sigma'_j]].$$

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

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

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

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

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

An example of a dynamically reconfigurable system is shown in Fig.1. This system consists of three DLHAs and one queue.

A DLHA $\mathcal{A}_1$ has two locations, *Run* and *Wait*, that are expressed as circles. In the location *Run*, an invariant is $x \leq 10$ and a flow condition is $\dot{x} = 1$.

A run of this system is as below:

$$\rho : \langle ((Run, x = 0), (Idle, y = 0), \perp), (\varepsilon) \rangle \quad (1)$$
$$\rightarrow^{10}_{q!\mathcal{A}_3} \langle ((Wait, x = 10), (Idle, y = 0), \perp), (\mathcal{A}_3) \rangle \quad (2)$$
$$\rightarrow^{0}_{q?\mathcal{A}_3} \langle ((Wait, x = 10), (Create, y = 0), \perp), (\varepsilon) \rangle \quad (3)$$
$$\rightarrow^{0}_{Crt\_\mathcal{A}_3} \langle ((Wait, x = 10), (Idle, y = 0), (Execute, z = 0)), (\varepsilon) \rangle \quad (4)$$
$$\rightarrow^{50}_{Dst\_\mathcal{A}_3} \langle ((Run, x = 0), (Idle, y = 0), \perp), (\varepsilon) \rangle \quad (5)$$
$$\rightarrow \cdots$$

Here, ε denotes the empty string.

In this tiny system, each DLHA has the following features:

- Environment: $\mathcal{A}_1$ periodically sends a message to request creation of $\mathcal{A}_3$.
- Dispatcher: $\mathcal{A}_2$ receives the message from $\mathcal{A}_1$ and creates $\mathcal{A}_3$ with actions *Crt_*$\mathcal{A}_3$! and *Crt_*$\mathcal{A}_3$?.
- Task: $\mathcal{A}_3$ is created by $\mathcal{A}_2$ synchronously with actions *Crt_*$\mathcal{A}_3$!.

In the initial state, $\mathcal{A}_1$ and $\mathcal{A}_2$ is already created because they have initial transitions with internal actions, and the content of the queue is empty (1). Next, $\mathcal{A}_1$ sends a creation request of $\mathcal{A}_3$ to $\mathcal{A}_2$ via the queue (2). Receiving the request asynchronously, $\mathcal{A}_2$ prepares to create $\mathcal{A}_3$, that is, $\mathcal{A}_2$ moves from *Idle* to *Create* and the content of the queue becomes empty (3). $\mathcal{A}_3$ is created and its state moves from $\perp$ to $(\textit{Execute}, z = 0)$ (4). When $\mathcal{A}_3$ finishes executing the process, it is destroyed with the action *Dst_*$\mathcal{A}_3$! (5).

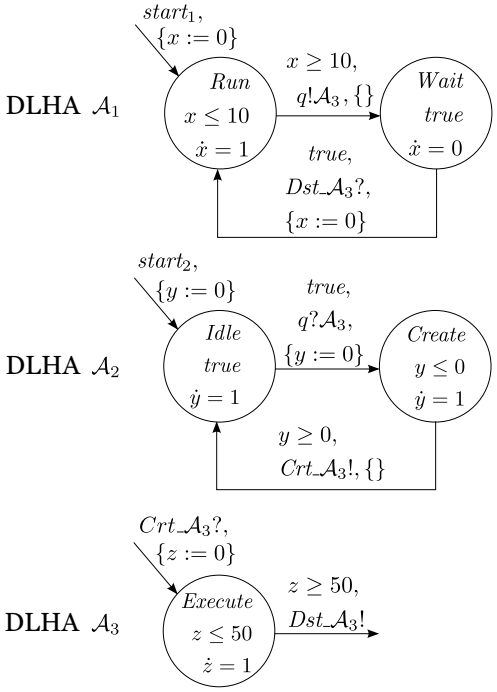

Fig. 1. An example of a dynamically reconfigurable system

IV. Reachability Analysis

A. Reachability Problem

Given a dynamically reconfigurable system $\mathcal{S} = (A, Q)$ and a location l_t, we say "$\mathcal{S}$ reaches l_t" if there exists a path from the initial state to a state containing l_t. The reachability problem is the problem of determining whether $\mathcal{S}$ reaches l_t.

B. Convex Polyhedra

In our proposed method, we introduce *convex polyhedra* for the reachability analysis according to [6]. For a set $V = \{x_1, \ldots, x_n\}$ of variables, a convex polyhedron ζ on V has the following syntax:

$$\zeta ::= \textit{true} \mid \textit{false} \mid \textstyle\sum_{i=1}^{n} a_i x_i \sim a_{n+1} \mid \zeta_1 \wedge \zeta_2$$

where $\sim \in \{=, <, >, \le, \ge\}$ and ζ_1, ζ_2 are convex polyhedra.

Let $[\![\zeta]\!]$ be a set of vertices such that are contained in the region described by ζ, that is, $[\![\zeta]\!] = \exists V.[\zeta] \subseteq \mathbb{R}^n$.

Then, the equivalence of convex polyhedra is defined as follows:

$$\zeta_1 = \zeta_2 \iff [\![\zeta_1]\!] = [\![\zeta_2]\!].$$

C. Algorithm

For the reachability analysis, a state of system is defined as $(L, \zeta, \vec{w_Q})$, where L is a set of locations, ζ is a convex polyhedron, and $\vec{w_Q}$ is a vector of queue-contents. The overview of the reachability analysis is shown in Fig. 2. The analysis is performed by breadth-first order with QDDs[7] as the following procedures:

1) Compute an initial state s_0 of the system $\mathcal{S}$ (line 1 – line 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, the following processing is repeated (line 8).
   a) Take a state $(L, \zeta, \vec{w_Q})$ from Wait and remove the state from Wait (line 9 – line 10).
   b) If the set L of locations contains the target location, return "yes" and terminate (line 11 – line 13).
   c) If the state is not traversed yet ($(L, \zeta, \vec{w_Q}) \notin$ Visit):
      i) Add the state into Visit (line 15).
      ii) Compute a set S_{post} of successors by the subroutine Succ (line 17).
      iii) Add all components of S_{post} into Wait (line 18).

The subroutine Succ that computes successors of a state is shown in Fig.3. Given a state $(L, \zeta, \vec{w_Q})$ and a system, successors are computed by the following procedure:

1) For each transition $(l, \phi, a, \lambda, l')$ (or destruction-transition $(l, \phi, a_l!)$) outgoing from a location $l \in L$, a set S_{post} of post states is computed as below:
2) If a is an enqueue action $q_k!w$: Add a message w to the content of queue q_k and add the new state to S_{post} (line 11 – line 15).
3) If a is a dequeue action $q_k?w$: Remove a message w from the content of queue q_k and add the new state to S_{post} (line 16 – line 19).
4) If a is other internal action: Add the new state to S_{post} (line 22).
5) If a is an output action $a_l!$: S_{post} is computed by subroutine Syncs as the following steps (line 29).
   a) Compute a set T_{sync} of transitions synchronizing with t.
   b) Compute a set L' of locations of the new state with t and T_{sync}.

Input: a system $\mathcal{S}$ and a target location l_t
Output: "yes" or "no"

/* Compute the initial state */
$L_0 \leftarrow \{l_{0i} \mid t_{0i} = (l_{0i}, a_{0i}, \lambda_{0i}), a_{0i} \neq \mathit{Crt\_}\mathcal{A}_i?\}$
$\lambda_0 \leftarrow \bigcup\{\lambda_{0_i} \mid t_{0i} = (l_{0i}, a_{0i}, \lambda_{0i}), a_{0i} \neq \mathit{Crt\_}\mathcal{A}_i?\}$
$s_0 \leftarrow (L_0, \vec{0}[\lambda_0], (\varepsilon, \ldots, \varepsilon))$
/* Initialize a set *Visit* of visited states and a queue *Wait* of states to search */
Visit $\leftarrow \varnothing$, Wait $\leftarrow \{s_0\}$
/* Traverse in a breadth-first order */
while Wait $\neq \varnothing$ **do**
    $(L, \zeta, \vec{w}_{\mathcal{Q}}) \leftarrow s \in$ Wait
    Wait $\leftarrow$ Wait $\setminus \{(L, \zeta, \vec{w}_{\mathcal{Q}})\}$
    if $l_t \in L$ **then return** "yes"
    if $(L, \zeta, \vec{w}_{\mathcal{Q}}) \notin$ Visit **then**
        Visit $\leftarrow$ Visit $\cup \{(L, \zeta, \vec{w}_{\mathcal{Q}})\}$
        /* Compute a set of post-states */
        $S_{post} \leftarrow$ Succ$((L, \zeta, \vec{w}_{\mathcal{Q}}), \mathcal{S})$
        Wait $\leftarrow$ Wait $\cup\, S_{post}$
return "no"

Fig. 2. Algorithm of Reachability Analysis

c) Compute a convex polyhedron ζ' of the new state with t and T_{sync}.
d) Add the new state $(L', \zeta', w_{\mathcal{Q}})$ to S_{post}.

6) If a is an input action: $S_{post} = \emptyset$.

V. Implementation and Practical Experiments

We have developed a prototype model checker of dynamically reconfigurable systems. The model checker is implemented in Java and comprised of about 1,600 lines of code using external libraries LAS[8], PPL[9], and QDD[7], [10].

Moreover, for practical experiments, we have specified a cooperating system consisting CPU and DRP(Dynamically Reconfigurable Processor), and verified several safety properties for the system. The configuration of the embedded system is shown in Fig.4, and components of the system is shown in Fig.5. This system consists of 11 DLHAs and 1 queue. We show the DLHA of *Scheduler* in Fig.6 for example. The external environment consists of EnvA and EnvB that create periodically TaskA and TaskB. That is, EnvA creates TaskA with *Crt_taskA*! every 70 milliseconds and EnvB creates TaskB with *Crt_taskB*! every 200 milliseconds. Scheduler 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, Scheduler receives *Crt_taskA?* and sends *Act_Preempt*!. Then, TaskA moves to location *RunA* and TaskB moves to the location *WaitB* with the action.

TaskA and TaskB send a message to Sender if they need a co-task. Sender enqueues the message to create a co-task to the queue q when it receives a message from tasks. When TaskA moves to location *RunA* from location *WaitA* with *Act_Create_a0*!, Sender receives *Act_Create_a0*? and enqueues *cotask_a0* to *q* with *q*!*cotask_a0*.

DRP_Dispatcher dequeues a message and creates co-task_a0, cotask_a1 and cotask_b0 if there are enough free tiles. Frequency_Manager is a module which manages the operating frequency of DRP. When a DLHA of co-task is created, Frequency_Manager moves to the location fixing the frequency to minimum value.

For example, we have verified the schedulability of tasks. This property provides that each task must be finished processing within its deadline. To verify the property, a *monitor automaton* is defined as Fig.7. Monitor automaton checks whether the system satisfies the property, and it moves to a special location called *error location* when the property is not satisfied. In this case, the schedulability is verified in 180 seconds with 169 [MB] on a machine with Intel (R) Core (TM) i7-3770 (3.40 [GHz]) CPU and 16 [GB] RAM.

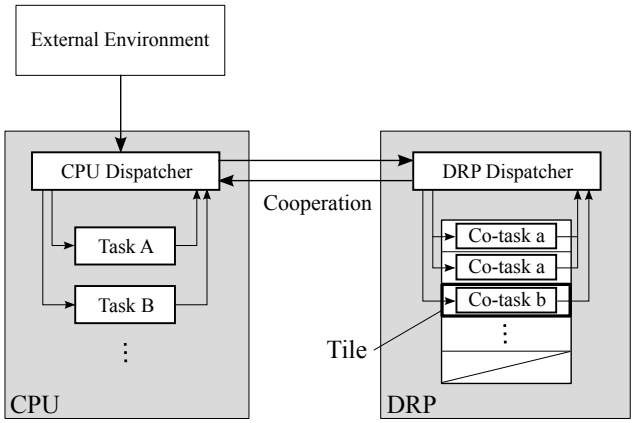

Fig. 4. Overview of the CPU-DRP embedded system

VI. Conclusion

In this paper, we have presented the specification language of dynamically reconfigurable systems and the reachability analysis algorithm for the verification. The next step would be to focus on more effective methods (e.g., Counterexample Guided Abstraction Refinement, Satisfiability Modulo Theories, etc.) for the verification.

Input: a state $(L, \zeta, \vec{w}_{\mathcal{Q}})$ and the system $\mathcal{S}$
Output: the set S_{post} of post-states

$T_N \leftarrow \bigcup_{i=1}^{|A|} \{(l, \phi_g, a, \lambda, l') \in T_i \mid l \in L\}$ /* Set of outgoing transitions */
$T_D \leftarrow \bigcup_{i=1}^{|A|} \{(l, \phi_g, a) \in T_{end_i} \mid l \in L\}$ /* Set of outgoing destruction transitions */
$S_{post} \leftarrow \emptyset$, $T_{post} \leftarrow T_N \cup T_D$
$\zeta_\delta \leftarrow \mathrm{Tsucc}(L, \zeta) \wedge \bigwedge_{l_p \in L} Inv_s(l_p)$ /* Convex polyhedron for the time transition */
for all $t \in T_{post}$ **do**
  if $t = (l, \phi_g, a, \lambda, l')$ **then**
    if a is an internal action **then**
      $L' \leftarrow (L \setminus \{l\}) \cup \{l'\}$ /* Locations of the post-state */
      $\zeta' \leftarrow (\zeta_\delta \wedge \phi_g)[\lambda] \wedge \zeta'_i \wedge \bigwedge_{l'_p \in L'} Inv_s(l'_p)$ /* The convex polyhedron of the post-states */
      if $\zeta' \neq$ *false* **then**
        if a is a enqueue action $q_k!w$ **then**
          $(w_1, \ldots, w_{|\mathcal{Q}|}) \leftarrow \vec{w}_{\mathcal{Q}}$
          $w'_k \leftarrow w_k w$ /* Enqueue the message into q_k */
          $\vec{w}'_{\mathcal{Q}} \leftarrow (w_1, \ldots, w_{k-1}, w'_k, w_{k+1}, \ldots, w_{|\mathcal{Q}|})$
          $S_{post} \leftarrow S_{post} \cup \{(L', \zeta', \vec{w}'_{\mathcal{Q}})\}$
        else if a is a dequeue action $q_k?w$ **then**
          if $w_k = ww'_k$ **then**
            $\vec{w}'_{\mathcal{Q}} \leftarrow (w_1, \ldots, w_{k-1}, w'_k, w_{k+1}, \ldots, w_{|\mathcal{Q}|})$ /* Dequeue the message from q_k */
            $S_{post} \leftarrow S_{post} \cup \{(L', \zeta', \vec{w}'_{\mathcal{Q}})\}$
          else
            $S_{post} \leftarrow S_{post} \cup \{(L', \zeta', \vec{w}_{\mathcal{Q}})\}$ /* For other internal action */
    else if a is an output action $a_l!$ **then**
      $S_{post} \leftarrow S_{post} \cup \mathrm{Syncs}((L, \zeta, \vec{w}_{\mathcal{Q}}), t, \mathcal{S})$ /* Compute the set of states by the synchronous transitions */
  else
    $S_{post} \leftarrow S_{post} \cup \mathrm{Syncs}((L, \zeta, \vec{w}_{\mathcal{Q}}), t, \mathcal{S})$ /* Compute the set of states by the destruction transition */
return S_{post}

Fig. 3. Subroutine Succ

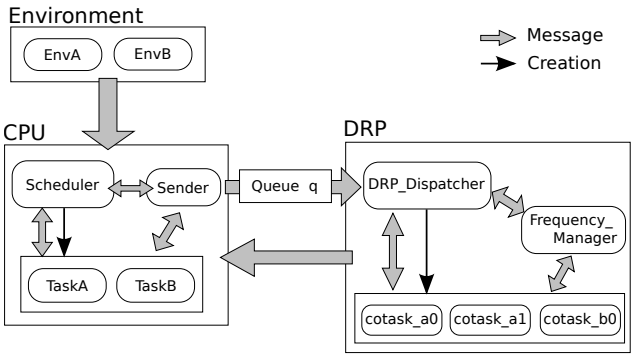

Fig. 5. Components of the system

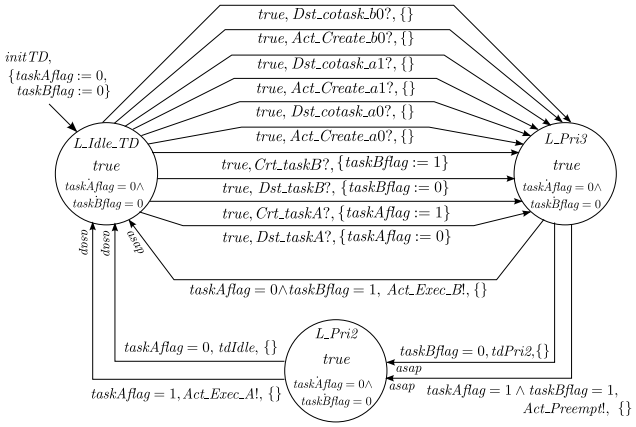

Fig. 6. DLHA of Scheduler

REFERENCES

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

[2] S. Minami, S. Takinai, S. Sekoguchi, Y. Nakai, and S. Yamane, "Modeling, specification and model checking of dynamically reconfigurable processors," in *Computer Software 28(1)*. Japan Society for Software Science and Technology, 2011, pp. 190–216.

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

[4] 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*, vol. 6, no. 3, pp. 1–19, 2013.

[5] H. Nakano, T. Shindo, T. Kazami, and M. Motomura, "Development of dynamically reconfigurable processor LSI," *NEC Technical Journal*, vol. 56, no. 2, pp. 99–102, 2003.

[6] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *THEORETICAL COMPUTER SCIENCE*, vol. 138, pp. 3–34, 1995.

[7] B. Boigelot and P. Godefroid, "Symbolic verification of communication protocols with infinite statespaces using qdds," *Form. Methods Syst. Des.*, vol. 14, no. 3, pp. 237–255, 1999.

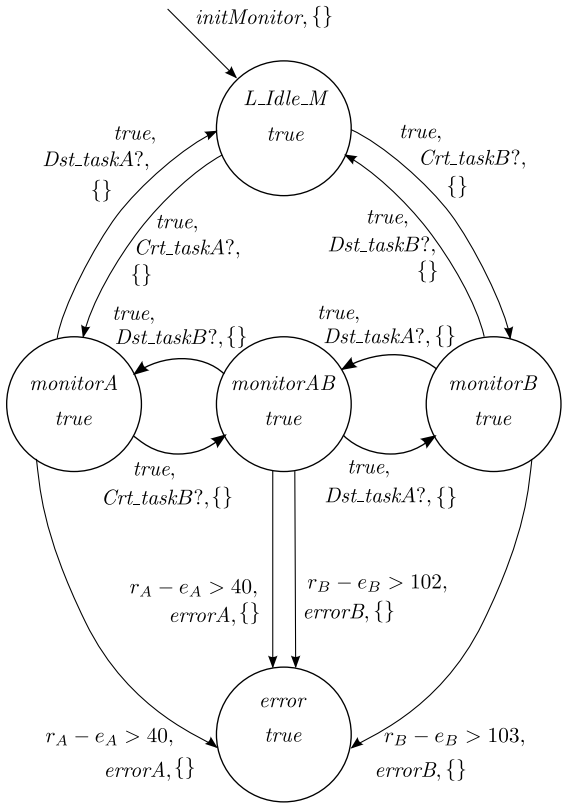

Fig. 7. Monitor automaton for schedulability

[8] Y. Ono and S. Yamane, "Computation of quantifier elimination of linear inequlities of first order predicate logic," *IEICE Technical Report. COMP, Computation*, vol. 111, no. 20, pp. 55–59, 2011.

[9] 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.*, vol. 72, no. 1-2, pp. 3–21, 2008.

[10] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper, "The power of qdds (extended abstract)," in *SAS*, 1997, pp. 172–186.