Queen's School of Computing

Technical Report

Timed Automata to Synthesize Controllers of Dynamic Hierarchical Real-Time Plants

Md Tawhid Bin Waez, Andrzej Wąsowski, Juergen Dingel, Karen Rudie

August 28, 2016

Timed Automata to Synthesize Controllers of Dynamic Hierarchical Real-Time Plants

Md Tawhid Bin Waez[1], Andrzej Wąsowski[2], Juergen Dingel[1], and Karen Rudie[1]

[1] Queen's University, Canada {waez@cs,dingel@cs,karen.rudie@}.queensu.ca
[2] IT University of Copenhagen, Denmark wasowski@itu.dk

Abstract. We propose *Timed Process Automata (TPA)* for modeling and analysis of time-critical systems which can be open, hierarchical, and dynamic. The model offers two essential features for large industrial systems: (i) compositional modeling with reusable designs for different contexts, and (ii) automated state-space reduction technique. Timed process automata model dynamic networks of continuous-time communicating control processes which can activate other processes. We show how to automatically establish safety and reachability properties of TPA by reduction to solving timed games. To mitigate the state-space explosion problem, an automated state-space reduction technique using compositional reasoning and aggressive abstractions is also proposed. We use timed game theory and Uppaal Tiga in a couple of industrial case studies and in the development of TPA. Both the case studies show that state-space explosion is a severe problem for timed games. Suitable abstractions, however, dramatically improve the scalability of timed games in one case study. These case studies motivate the development of TPA and an automatable state-space reduction technique for them based on aggressive abstraction.

1 Introduction

Automata are a prominent group of models in model-based development because they facilitate many important types of formal analyses. *Finite automata* (and their derived models, such as *Kripke structures* [1]) can be considered as the most popular, studied, and applied automata because of their rich theoretical properties and practicability. Properties of some systems, however, do not depend only on exact sequence of actions (or communication) but also the exact *time* of execution. Finite automata, implicitly, can model time information using sample timed data. For example, an action a that executes n seconds after the previous action b can be modeled as n special time tick symbols followed by a. Such implicit modeling of time can result in an exponential blowup of both input data and the size of the model. To avoid this problem, this paper uses *timed automata (TA)* [2,3], which can be viewed as finite automata with continuous clocks to record time. Timed automata are preferred to other real-time formal models (such as timed Petri nets [4], timed transition systems [5], and Modecharts) because real-time reachability and some other important analytical properties

were first solved using symbolic semantics *region graph* of TA and after that other models adopted the same approach.

An *open system* continuously interacts with an unpredictable environment. A *hierarchical system* is a hierarchical composition of smaller systems. An automotive system, developed by an *original equipment manufacturer (OEM)*, may be used in different models of cars. In this case, the system has a *controller* which helps the system adapt to different *environments* and cars. In other words, the system is an open system, which has two distinguished interacting segments: the controller and the environment. Typically, these systems consist of other smaller systems in a *hierarchical* structure. For instance, a system Actuator can be a component of a larger system Position, while Position can be a component of another system Brake-by-Wire, and so on. Every component of a system has a specific set of tasks; for example, system Brake-by-Wire may use its component Position to perform some desired tasks in interaction with the environment, and Brake-by-Wire may also indirectly—through using Position—use its sub-component Actuator to perform some desired tasks in interaction with the environment.

A *dynamic hierarchical system* is a hierarchical system whose components may change over time. Many hierarchical systems have dynamic characteristics, which are activating components only when needed. Dynamic behaviors are an important feature when resource constraints (such as limited memory) do not allow one to keep all the components active at the same time. Sometimes dynamic behaviors are inherent to the system. For example, we applied timed game theory in an industrial project to construct a fault-tolerant framework for a hierarchical open system that has a scheduler, a set of tasks, and a set of subtasks; only the scheduler is active in the initial system-state; subtasks are activated by their parent tasks, and the top level tasks are activated by their scheduler; thus the scheduler controls tasks, and a task controls its subtasks; due to the termination or the initialization of tasks (or subtasks) the structures of the processes may change; thus the system is a dynamic open system [6].

A *ground hierarchical open system* is a hierarchical (open) system that does not have a component. A non-ground hierarchical open system is a *compound hierarchical open system*. A ground hierarchical open system has a control hierarchy of depth 0. A compound hierarchical open system system_1 has a control hierarchy of depth $n+1$, where n is the maximum of depths of the control hierarchies of the components contained in system_1. Many hierarchical open systems have *dynamic behaviors*, which have components that are activated only when needed. Dynamic behaviors are an important feature when resource constraints (such as limited memory) do not allow one to keep all the components active at the same time. Models of industrial dynamic hierarchical open systems can be very detailed because of the hierarchical compositionalability. These details may introduce errors in the design and make automated analysis challenging.

Timed automata are desirable for the development of *open real-time systems* since TA can capture both discrete-time controllable behaviors of the system and dense-time uncontrollable behaviors of the environment. Timed automata have no explicit structured support for modeling dynamic hierarchical open systems. This

absence may lead to cumbersome design details in a large-scale system having several *control hierarchies*. *Timed game automata* [7,8,9]—a variant of TA—are a well-known model in the research community for the analysis of open dense-time systems. Dense-time formal methods of TA may provide the most accurate analysis, however TA, currently, are not suited for open systems in practice mainly because of poor scalability. This paper proposes the first compositional modeling with reuse and automatable-state-space reduction technique for the formal analysis of *dynamic hierarchical open real-time systems*. This paper applies TA to an industrial problem and proposes a novel variant TA together with a state-space reduction technique for the compositional modeling and analysis of dynamic hierarchical open real-time systems.

1.1 Background

A timed automaton is a finite state automaton with a set of asynchronous nonnegative real valued *clocks* and a set of clock constraints. If a timed automaton is considered as a directed graph, locations represent the vertices of the graph, and locations are connected by edges. Locations of a timed automaton are graphically represented as circles. A *clock valuation* over the set of clocks is a mapping which assigns to each clock a nonnegative real value. An *initial clock valuation* maps each clock of a timed automaton to zero. The clock constraint which is associated with a *location* is called the *local invariant* of that location. Control can stay in a location only if the clock valuation satisfies the local invariant of that location. Local invariants are used to ensure the progress of the model [10], that is, control can stay in a location until its local invariant permits. An edge in a timed automaton is associated with a clock constraint, a subset of the clocks, and a *label*. The clock constraint which is associated with an edge is called the *guard* of that edge. An edge can be traversed only if the clock valuation satisfies the guard of that edge. Clock constraints are used to restrict the timing behaviors of the automaton. Each associated clock of an edge is reset to 0 when the edge traverses. At any instant, the value of a clock equals the time elapsed since the last time it was reset. While edges are instantaneous, time can elapse in a location. The semantic construction of TA is expressed using semantics objects called *timed transition systems* [11,9,3]. A *timed I/O automaton* [12,13,9] is a timed automaton which has an input alphabet along with a regular output alphabet. The controller plays controllable output transitions and the environment plays uncontrollable input transitions; thus timed I/O automata (TIOA) are a natural model for timed games. Two TIOA are *composable* with each other if they do not have a common output action.

Definition 1 *[11,9,3] A* timed transition system *(with only one initial location but without final location and ϵ-transition) is a tuple $\mathcal{T} = (St, s_0, \Sigma, \dashrightarrow)$, where St is a set of* states*, $s_0 \in St$ is the* initial state*, Σ is an alphabet, and $\dashrightarrow: St \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times St$ is a* transition relation.

We use $d \in \mathbb{R}_{\geq 0}$ to denote delay. A timed transition system satisfies *time determinism* (i.e., whenever $s \overset{d}{\dashrightarrow} s'$ and $s \overset{d}{\dashrightarrow} s''$ then $s' = s''$ for all $s \in S$), *time*

reflexivity (i.e., $s \overset{0}{\dashrightarrow} s$ for all $s \in S$), and *time additivity* (i.e., for all $s, s'' \in S$ and all $d_1, d_2 \in \mathbb{R}_{\geq 0}$ we have $s \overset{d_1+d_2}{\dashrightarrow} s''$ iff there exists an s' such that $s \overset{d_1}{\dashrightarrow} s'$ and $s' \overset{d_2}{\dashrightarrow} s''$). A *run* ρ of a timed transition system $\mathcal{T}$ from a state $s_1 \in St$ is a sequence $s_1 \overset{a_1}{\dashrightarrow} s_2 \overset{a_2}{\dashrightarrow} s_3 \cdots \overset{a_n}{\dashrightarrow} s_{n+1}$ such that for all $1 \leq m \leq n : s_m \overset{a_m}{\dashrightarrow} s_{m+1}$ with $a_m \in \Sigma \cup \mathbb{R}_{\geq 0}$. A state s is *reachable* in a transition system $\mathcal{T}$ if and only if there is a run $s_0 \overset{a_0}{\dashrightarrow} s_1 \overset{a_1}{\dashrightarrow} s_2 \cdots \overset{a_{n-1}}{\dashrightarrow} s_n$, where $s = s_n$. *Timed I/O transition systems* are timed transition system with input and output modalities on transitions. Timed I/O transition systems are used to define semantics of TIOA. A *constraint* $\delta \in C(X, V)$ over a set of clocks X and over a set *counters*, non-negative finitely bounded integer variables, V is generated by the grammar $\delta ::= x_m \prec q \mid k \prec \alpha \mid x_m - x_n \prec q \mid true \mid \Phi \wedge \Phi$, where $q \in \mathbb{Q}_{\geq 0}$, $\alpha \in \mathbb{Z}_{\geq 0}$, $\{x_m, x_n\} \subseteq X$, $k \in V$ and $\prec \in \{<, \leq, >, \geq\}$. Consequently, the set of *clock constraints* $C(X)$ is the set of constraints $C(X, V)$, where $V = \emptyset$. Let $\Psi(V)$ be the set of assignments over the set of variables V.

Definition 2 *[12,13,9,3] A* timed I/O automaton *is a tuple $\mathcal{A} = (L, l_0, X, V, A, E, I)$, where L is a finite set of locations, $l_0 \in L$ is the initial location, X is a finite set of clocks, V is a finite set of counters, $A = A_{\mathrm{i}} \oplus A_{\mathrm{o}}$ is a finite set of actions, partitioned into input actions A_{i} and output actions A_{o}, $E \subseteq L \times A \times \Phi(X, V) \times \Psi(V) \times 2^X \times L$ is a set of edges, and $I : L \to C(X)$ is a total mapping from locations to invariants.*

A *clock valuation* over X is a mapping $\mathbb{R}^X_{\geq 0} : X \to \mathbb{R}_{\geq 0}$ and a *counter valuation* over V is a mapping $\mathbb{Z}^V_{\geq 0} : V \to \mathbb{Z}_{\geq 0}$. Given a clock valuation $v \in \mathbb{R}^X_{\geq 0}$ and $d \in \mathbb{R}_{\geq 0}$, we write $v + d$ for the clock valuation in which for each clock $x \in X$ we have $(v + d)(x) = v(x) + d$. For $\lambda \subseteq X$, we write $v[x \mapsto 0]_{x \in \lambda}$ for a clock valuation agreeing with v on clocks in $X \setminus \lambda$, and giving 0 for clocks in λ. For $\phi \in \Phi(X, V)$, $v \in \mathbb{R}^X_{\geq 0}$, and $n \in \mathbb{Z}^V_{\geq 0}$, we write $v, n \models \phi$ if v and n satisfy ϕ. Let $e = (l, a, \phi, \theta, \lambda, l')$ be an edge, then l is the source location, a is the action label, and l' is the target location of e; the constraint ϕ has to be satisfied during the traversal of e; the set of clocks $\lambda \in 2^X$ are reset to 0 and the set of counters are updated to θ whenever e is traversed.

Definition 3 *[13,9] Two TIOA $\mathcal{A}^m = (L^m, l^m_0, X^m, V^m, A^m, E^m, I^m)$ and $\mathcal{A}^n = (L^n, l^n_0, X^n, V^n, A^n, E^n, I^n)$ are* composable *with each other when $A^m_o \cap A^n_o = \emptyset$, $X^m \cap X^n = \emptyset$, and $V^m \cap V^n = \emptyset$; when composable, their* composition *is a TIOA $\mathcal{A} = \mathcal{A}^m \| \mathcal{A}^n = (L^m \times L^n, (l^m_0, l^n_0), X^m \cup X^n, V^m \cup V^n, A, E, I)$, where $A = A_{\mathrm{i}} \cup A_{\mathrm{o}}$ with $A_{\mathrm{o}} = A^m_{\mathrm{o}} \cup A^n_{\mathrm{o}}$ and $A_{\mathrm{i}} = (A^m_{\mathrm{i}} \cup A^n_{\mathrm{i}}) \setminus A_{\mathrm{o}}$. The set of edges E contains:*

- $((l^m, l^n), a, \phi^m \wedge \phi^n, \lambda^m \cup \lambda^n, \theta^m \cup \theta^n, (l'^m, l'^n)) \in E$ *for each* $(l^m, a, \phi^m, \theta^m, \lambda^m, l'^m) \in E^m$ *and* $(l^n, a, \phi^n, \theta^n, \lambda^n, l'^n) \in E^n$ *if* $a \in \{A^m_{\mathrm{i}} \cap A^n_{\mathrm{o}}\} \cup \{A^m_{\mathrm{o}} \cap A^n_{\mathrm{i}}\}$
- $((l^m, l^n), a, \phi^m, \lambda^m, \theta^m, (l'^m, l^n)) \in E$ *for each* $(l^m, a, \phi^m, \lambda^m, \theta^m, l'^m) \in E^m$ *if* $a \notin A^n$
- $((l^m, l^n), a, \phi^n, \lambda^n, \theta^n, (l^m, l'^n)) \in E$ *for each* $(l^n, a, \phi^n, \lambda^n, \theta^n, l'^n) \in E^n$ *if* $a \notin A^m$

and the set of invariants I is constructed as follows: $I(l^m, l^n) = I^m(l^m) \wedge I^n(l^n)$

A real-time control problem can be viewed as a two-player timed game [7,14,15] between the controller and the environment, where the controller aims to find a

strategy to guarantee that the system will satisfy a given property, no matter what the environment does [16]. An example of such reformulation is to find a strategy for the controller (or a reconfiguration service) to prevent the system from becoming unstable in the presence of the faults of the fault model. The *game reachability* problem is whether the system has a strategy to reach a target state regardless of how the environment behaves. The *game minimum-time reachability* problem in timed game automata is finding the minimum time required by the system to reach a target state regardless of how the environment behaves. Uppaal Tiga [17], a TIOA-based tool, is a tool for solving games based on timed game automata with respect to reachability and safety properties. Synthia [18] performs verification and controller synthesis for timed games.

1.2 Motivation

The first goal of this paper is to develop an automatable synthesis technique for reconfiguration services for cost-effective fault tolerance. The next goal is to develop a TA-based modeling paradigm for dynamic hierarchical open systems, where a designer will not need to readjust a design for different compositions. However, the main motivation is to develop an automatable state-space reduction technique for TA-based analysis of dynamic hierarchical open systems.

Industrial multi-core systems typically use additional processing cores to provide fault-tolerance. Task-level reconfiguration techniques reduce the number of these additional processing cores—thus reducing costs—by reallocating the loads of the failed cores to the non-additional operational cores. The main challenge for developing a reconfiguration technique is to provide formal guarantee that the developed technique can handle all fault scenarios. Automated formal synthesis of such reconfiguration frameworks is highly desirable for industrial use.

Figure 1 presents an abstract Brake-by-Wire system modeled using TIOA, and the system is developed by an OEM. The model has seven automata representing different copies of only three elements: one copy of the *main thread* of Brake-by-Wire (the top automaton), two copies of the main thread of Position (the two automata in the middle), and four copies of Actuator system (the four automata in the bottom). Each Position system contains two *children* (Actuator systems) and its main thread that schedules the children, communicates with its parent (the main thread of Brake-by-Wire), and performs some other functions, which cannot be performed by the children. Similarly, this Brake-by-Wire system contains two children (Position systems) and its main thread that schedules the children and performs some other functions, which cannot be performed by the children. In this model, the main thread of Brake-by-Wire is the *root*, which does not have a parent. However, in the future a car manufacturer may include this Brake-by-Wire system in a car and then the main thread of Brake-by-Wire will no longer be the root. Then a central control system may be able to start the main thread of Brake-by-Wire. To analyze the new complex system, a designer will need to manually alter the model again by including *start* and *finish* actions (in the top automaton of Figure 1). Let us assume a complex system contains N Break-by-Wire systems; to analyze this complex system, a designer will need

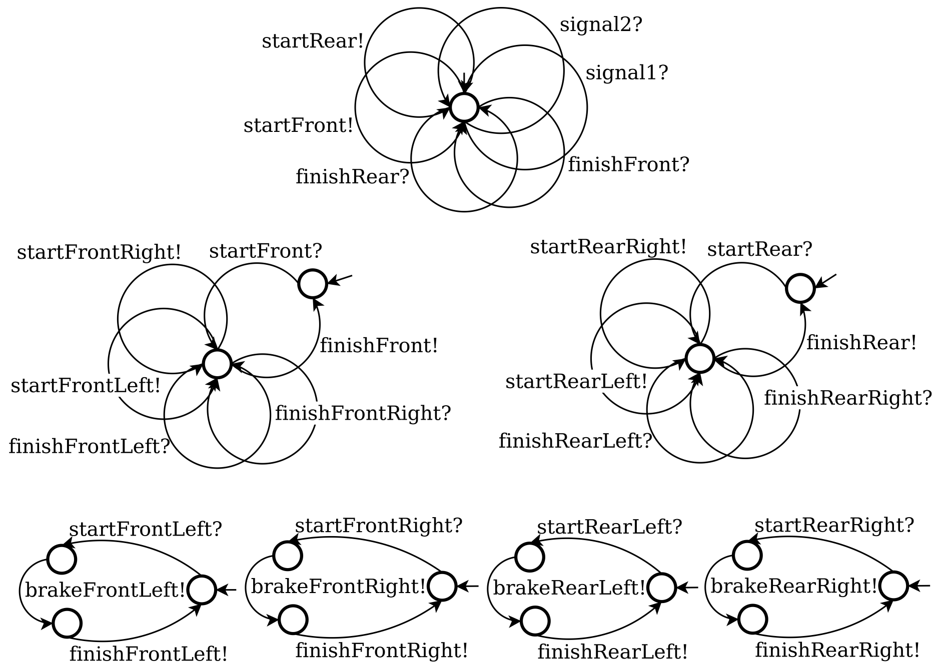


Fig. 1. An abstract Brake-by-Wire system modeled using standard TIOA, where one copy of the *main thread* of Brake-by-Wire (the top automaton), two copies of the main thread of Position (the two automata in the middle), and four copies of Actuator system (the four automata in the bottom)

to manually construct at least $N \times 7$ automata with a proportionally growing alphabet! Existing TA-based modeling techniques do not support compositional modeling with reusable designs for different contexts; that is, a design may need to be altered manually in every composition. All these ad hoc alterations may make a large industrial design incomprehensible and error-prone.

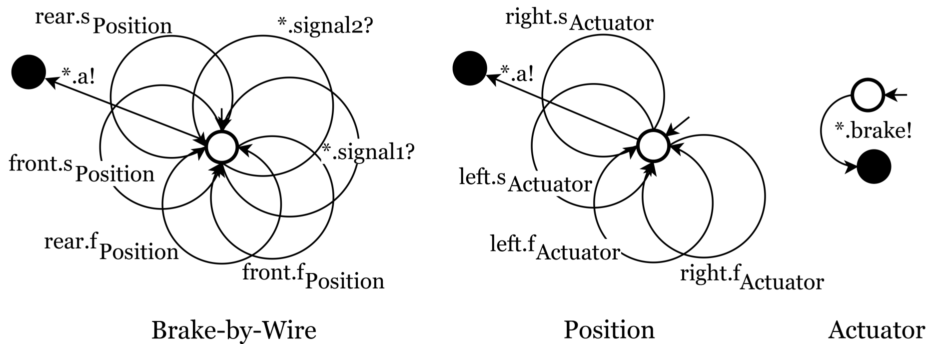


Fig. 2. The same Brake-by-Wire system of Figure 1 is modeled using TPA

Figure 2 contains the same Brake-by-Wire system of Figure 1 modeled by using our proposed *timed process automata (TPA)*, which always model a system only once. For example, Figure 2 presents only three TPA, which are equivalent to the seven automata of Figure 1. Moreover, the number of copies and the root status of Break-by-Wire system has no impact on the new design.

No automated state-space reduction technique has been developed for the analysis of dynamic hierarchical open dense-time systems. During our two projects with an automotive manufacturer, we noticed that even a (practically) very small real-time system may have a state space too large for automated formal analysis because of hierarchy, dynamic behaviors, and time calculations. We overcame the scalability problem in one of the projects—construction of a fault-tolerance framework in Section 2 and in [6]—by developing a manual state-space reduction technique that applies aggressive abstractions and uses fewer synchronizations. Applying this manual technique to a design of an industrial system is a challenging task. A generalized automated reduction technique, therefore, is needed for analysis of dynamic hierarchical open time-critical systems, which is provided by presenting an automatable reduction technique for TPA.

1.3 Problem

The problem we address is to develop an automatable synthesis technique for reconfiguration frameworks using TA, and develop a theoretical foundation for TA 1) to allow compositional modeling with reuse for dynamic hierarchical open systems and 2) to allow timed games-based automatable analysis for large dynamic hierarchical open systems.

Challenges The main challenge is timed games-based synthesis has poor scalability. Moreover, for hierarchical compositional systems, the size of the composition in the monolithic analysis is exponential in the depth of the hierarchy of the system due to the product construction of the state space. Furthermore, the state space in the analysis is also linear in the product of the sizes of all included components of the system. The components of industrial hierarchical systems, unfortunately, typically are very detailed. For automatable reuse in compositional modeling, we need automatable techniques 1) to convert an independent system into a component of a larger system and 2) to construct n copies of component C of system system_1 (such as $C_1, C_2, \cdots C_n$) in a way that these new copies can communicate with the other components of system_1 and the environment.

Scope Engineers of our industrial collaborator aimed to develop service-based solution for task-level reconfigurations to achieve fault tolerance for mixed-criticality multi-core systems. However, they were struggling to provide formal guarantee that the proposed services ensure fault tolerance. This paper synthesizes these services with formal assurance to solve their problem. Moreover, this sysntheis process can be automated. Theoretically, TPA are not more expressive than timed game automata. For instance, on the semantic level TPA use timed

games for the analysis. However, TPA allow automatable analysis of larger dynamic hierarchical open systems. Timed process automata allow automated controlled safety and reachability analysis of arbitrary number of processes; but there is an implicit bound on the maximal number of active processes at a time.

Organization The paper concludes in Section 4 before that it provides:

Section 2 A novel automatable synthesis technique for reconfiguration services that assures fault tolerance of mixed-criticality multi-core systems.
Section 2.5 Results of experiments provide evidence of the usefulness of aggressive abstractions for state-space reduction.
Section 3.1 A TA variant called timed process automata that provides compositionality with reuse feature to model dynamic hierarchical open systems.
Section 3.2 An automated dense-time controllability analysis technique for the developed model.
Section 3.3 An automatable state-space reduction technique for the developed automated controllability analysis, which will allow the analyses of larger dynamic hierarchical open systems.
Section 3.4 Results of experiments to determine effectiveness of the developed state-space reduction technique. The result provides evidence of the usefulness of the technique.

1.4 Discussion

We propose TPA in Section 3. Resource constraints may not permit a hierarchical system to activate all of its components at the same time. Such resource constraints of can be modeled using TPA thus they are a variant of TA with resources [19]. Task automata [20,19] can model only two layers of hierarchy and only closed systems. Our proposed variant can model any numbers of hierarchies and can model both closed and open systems. Moreover, TPA can model private communication among components. Timed process automata are a member of the class of TA with succinctness [19]—such as *TA with deadlines* [21]— because they hide many design details from the designers to achieve succinctness. Timed process automata are also timed game automata [7,15,13,9] because the new variant uses timed games for analysis.

2 Synthesis of a Reconfiguration Service

We synthesize task-level reconfiguration services to ensure fault-tolerance of a mixed-criticality automotive system that consists of an asymmetric multi-core processor (AMP). The system has a fault-intolerant AMP scheduler. We augment the existing scheduler with supplementary reconfiguration services, which we synthesize. The services assure the periodic executions of all the critical tasks in the presence of faults from a fault model.

We use timed games at synthesis-time and lookup tables at runtime to provide task-level reconfiguration, a cost-effective fault-tolerance technique, for mixed-criticality multi-core systems. System-level requirements for embedded, real-time software in many domains (such as automotive) have enough flexibility to reallocate tasks from one processing core to another. A task-level reconfiguration technique reduces the number of redundant cores that are used only to provide fault-tolerance by reallocating the loads of the failed cores to the non-redundant operational cores. Reduction in the amount of expensive hardware gives task-level reconfiguration a hope to be a dominant fault-tolerance technique in the automotive industry, where cost-efficiency and fault-tolerance are both crucial issues. Since this economical technique can handle tasks with different levels of criticality, one of its prospective application sectors is next-generation automotive systems, most of which are expected to be mixed-criticality multi-core systems.

Formal methods have been used for the development of fault-tolerant real-time systems. However, in the industry, fault-tolerance problems are typically designed, analyzed, and solved using classical control theory [22,23]. Timed game theory [7,14,15], a dense-time automata-based game theory, is almost unexplored in the industry. The use of timed game theory to solve industrial problems is attractive because of automated controller synthesis, visual modeling, and dense-time formal analysis. Nevertheless, applying timed game theory to solve industrial problems is challenging because of its high computational complexity.

We use timed games to synthesize the embedded controllers of the reconfiguration services. Our approach guarantees fault-tolerance up to a certain maximal number of concurrent faults after inserting the services into the system. Such reliable and accurate information is very helpful to build mixed-criticality systems cost effectively. Intellectual property regulations do not allow us to present the case study on the systems of our industry partner. Instead we demonstrate the synthesis process using a small system, which is complex enough to show the essence of the problem and our approach, yet simple enough to allow a compact and comprehensible presentation.

Dense-time models can capture fault occurrences and other uncontrollable behaviors at dense-time, not only uncontrollable behaviors at discrete time. Timed automata have been used for many purposes [19] including for fault diagnosis [24,25,26], analyzing multi-core systems [27,28], task models [29], and analyzing mixed-criticality systems [30].

2.1 Systems

We consider a class of multi-core systems having asymmetric processing cores. Different asymmetric cores may exhibit different performance for the same task. The systems under consideration are mixed-criticality systems, because they execute both critical tasks and non-critical tasks with two different priorities.

Definition 4 *A mixed-criticality system, of our consideration, consists of*

- *N asymmetric processing cores:* core_1, core_2, $\cdots$, core_N

- *M tasks:* task_1*,* task_2*,* $\cdots$*,* task_M
- *P critical tasks, where $P < M$*
- *A fault-intolerant criticality-unaware AMP scheduler with a static allocation of tasks*
- `load`$(\mathsf{task}_i, \mathsf{core}_j)$ *is a function mapping each task-core pair to the worst-case load that the task generates on the core, represented as a number* $\{0, 1, \cdots, 100\} \cup \{+\infty\}$*, where* $+\infty$ *represents incompatibility between the core and the task.*
- *Function* `primary`(task_i) *maps* task_i *to the core on which the task runs in the initial system-state*
- *Predicate* `critical`(task_i) *holds only for critical tasks*
- *Each task is executed periodically. Tasks always terminate within the prescribed periods. Each task is described as a TIOA. These automata do not communicate*[3]*. Every task can be killed (and resumed) in any of its states by a reconfiguration technique.*
- *Fault Model: The system is fault-free in its initial system-state. In the other system-states, the system might suffer three types of faults: safety violations by tasks, permanent core failures, and temporary core failures. Critical tasks are assumed not to breach any safety constraints. Non-critical tasks may violate safety constraints. Every core of the system may fail. However, all cores of a system cannot simultaneously be in their failed states. The maximal number of cores that can fail concurrently is restricted by CFL, concurrent-failures–limit. No limit is imposed on the total number of fault occurrences in a run.*

Given a mixed-criticality system of Definition 4, we want to obtain a task allocation policy that is able to cope with the failures admitted by the fault model. We will synthesize distributed reconfiguration services that assure uninterrupted executions of all the critical tasks. Section 2.2 explains how the reconfiguration technique is expected to work using an example.

2.2 Task-Level Reconfiguration Service

We propose a service-based reconfiguration technique for the fault-tolerance of mixed-criticality systems, where the system has a task-level reconfiguration service for each core. The services manage critical tasks differently from non-critical tasks. Consider, for instance, a simple mixed-criticality AMP system system_1, one of the systems that are described in Section 2.1. System system_1 executes six periodic tasks `S`, `W`, `D`, $\mathtt{N}_1$, $\mathtt{N}_2$, and $\mathtt{N}_3$. Only three tasks `S`, `W`, and `D` are the critical tasks, where in an execution `S` records exactly one update of a speedometer, and `W` (respectively, `D`) records at most one update of a wiper (resp., door). The system has three cores core_1, core_2, and core_3, which are asymmetric but each core is able to execute all six tasks.

[3] More generally, the communication can be abstracted by suitable understanding of worst and best case execution times, and terminations are independent of communication

Figure 3 presents a trace of a desirable behavior of system_1 in the presence of different faults after inserting the reconfiguration services; the figure omits suspended non-critical tasks to avoid clutter. At any given time, the periodic execution of a task can be assigned to at most one operational core. A task is assigned to its primary core in the initial system-state, where a core is responsible to execute only its primary tasks. For instance, core_1 is the primary core of task S, and S is a primary task of core_1 in Figure 3. We call a non-primary core a backup core of a critical task when that core can execute that task; similarly, a task is a backup task of its backup core. Whenever a core fails, the services assign the critical tasks that were previously assigned to that failed core to the operational cores. The services may kill and suspend temporarily one or more non-critical tasks on the operational cores during a reallocation process to ensure enough processing capacity for the reallocated critical tasks. In Figure 3, core core_2 fails in system-state s_2; in the next system-state, the periodic execution of critical task W is assigned to a backup core core_3 and the periodic execution of non-critical task $\mathtt{N}_3$ is suspended temporarily on core_3 to have enough processing capacity for W. A critical task is allowed to execute further on a backup core only if the primary core is in a failed state. The services kill a critical task on a backup core (if that task is initialized or released) and cancels the assignment of that task on that backup core, whenever the primary core recovers from a temporary failure. As an example, core core_2 recovers from a temporary failure in system-state s_6, and after that only core_2 is assigned to perform critical task W. The services reinstate a suspended non-critical task as soon as enough processing capacity for that task is regained due to the recovery of a core from a temporary failure; for example, the periodic execution of non-critical task $\mathtt{N}_3$ is reinstated in system-state s_7. The services permanently suspend a non-critical task when it performs some harmful activities, such as illegal memory access. For instance,

s_1	$core_1$: operational S : primary N_1: safe	$core_2$: operational W: primary N_2: safe	$core_3$: operational D: primary N_3: safe
s_2	$core_1$: operational S: primary N_1: safe	$core_2$: failed W: primary N_2: safe	$core_3$: operational D: primary N_3: safe
s_3	$core_1$: operational S: primary N_1: safe	$core_2$: failed	$core_3$: operational D: primary W: backup
s_4	$core_1$: operational S: primary N_1: unsafe	$core_2$: failed	$core_3$: operational D: primary W: backup
s_5	$core_1$: operational S: primary	$core_2$: failed	$core_3$: operational D: primary W: backup
s_6	$core_1$: operational S: primary	$core_2$: operational	$core_3$: operational D: primary W: backup
s_7	$core_1$: operational S: primary	$core_2$: operational W: primary N_2: safe	$core_3$: operational D: primary N_3: safe

Fig. 3. Sample trace of system_1 with reconfiguration

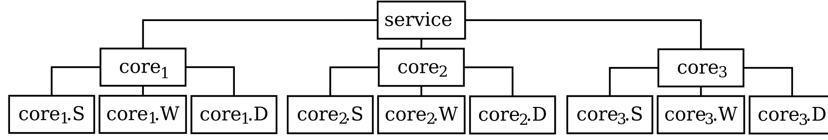


Fig. 4. Architecture of system_1 after adapting abstractions of Section 2.3

non-critical task $\mathtt{N}_1$ performs some harmful activities in system-state s_4, and the task is permanently suspended in system-state s_5.

Problem Statement Given a mixed-criticality system as specified in Definition 4, the problem is to synthesize a reconfiguration service $\mathsf{service}_i$ for each core core_i such that $\mathsf{service}_i$: (i) reacts whenever any other core fails or a core recovers (including core_i), or a non-critical task violates a safety constraint on core_i; (ii) at that time $\mathsf{service}_i$ may kill, resume, and suspend any task running on core_i; and (iii) as long as core_i is in a failure state, none of its tasks nor $\mathsf{service}_i$ executes. All reconfiguration services of a system together satisfy a property that at all times critical tasks are allocated to operating cores as long as the CFL limit is observed, and any non-critical task that has violated a safety constraint is suspended from execution indefinitely.

2.3 Modeling

We construct a timed game model of the system in a way that an unsafe location becomes reachable when a core exceeds its processing capacity. The model explicitly or implicitly captures the behaviors of the scheduler, the reconfiguration services, the cores, and the tasks.

To reduce complexity: (i) we model only a single (central) reconfiguration service for the whole system, instead of one service per core; (ii) we assume that every non-idle state of a task requires the worst-case core load of the task on the current core; and (iii) we abstract away the non-critical tasks. These three assumptions do not prevent synthesis of a distributed reconfiguration service per core, which will be shown in Section 2.4. Our model depends on four system parameters: (i) the release period of each task (constants `pS`, `pW`, `pD`); (ii) the worst-case load of each task on each core, in percent of the processing capacity of that core (constants `lS1`, `lW1`, `lD1`, `lS2`, `lW2`, `lD2`, `lS3`, `lW3`, `lD3`); (iii) the worst-case execution time (WCET) of each task on all cores (constants `wS`, `wW`, `wD`); and (iv) the best-case execution time (BCET) of each task on all cores (constants `bS`, `bW`, `bD`).

Now we illustrate the design process by constructing a concrete model of mixed-criticality AMP system system_1. The main design principle behind this model is to describe each component of the system in detail as a TIOA then obtain an intuitive concrete model by composing all the components using parallel composition [9]. The concrete model has 13 TIOA, which follow five different templates. In general, the concrete model has at most $(N \times K) + N + 1$ TIOA and

$K + 2$ templates, where N is the number of total cores, K is the number of total critical tasks, constant 1 automaton for the central service, constant 1 template for cores, and constant 1 template for the central service.

Each automaton of the concrete model represents exactly one rectangle of Figure 4. The automata synchronize using both actions and global variables. The model does not have any local variables and constants. A task automaton models initialization, killing, resumption, termination, and state information of a task on a specific core; for example, task automaton core_1.S in the bottom of Figure 5 represents the activities of task S on core_1. A core may fail only if the fault model allows it to fail. A core automaton models initializations and terminations of tasks on a core along with failures of the core and safety violations; for instance, core automaton core_1 in the middle of Figure 5 represents the activities of core core_1. The `service` automaton in the top of Figure 5 models reallocations of the critical tasks when a core fails or recovers. In the model a failed core may recover at any time. All automata of the model are presented in the appendix.

The automata modeling cores follow the same template. For instance, automaton core_1 uses action `kS1` to model the killing of task S on core_1 (edge 16 in Figure 5), `kW1` to model the killing of task W on core_1 (edge 17), `kD1` to model the killing of task D on core_1 (edge 18), and global variable `L1` to record the current worst-case load on core_1 (edges 9–14,16–18); similarly, automaton core_2 uses action `kS2` to model the killing of S on core_2 and global variable `L2` to record the current worst-case load on core_2. The automata modeling the same task—but on different cores—follow the same template.

Task Automata A task automaton represents two types of activities of a task on a core:

Task Life-Cycle Activities (edges 1–5) Every task can be initialized, killed, and resumed by performing controllable actions. Task terminations are modeled using uncontrollable actions because neither the scheduler nor the reconfiguration services can control the exact termination period of a task. The models are built in Uppaal Tiga [17], which displays controllable transitions as solid arrows (edges 1–4), and uncontrollable transitions as dashed arrows (edges 5–8). The duration between an initialization and the immediate termination of a task encompasses one complete execution of that task. A task can be killed and then resumed arbitrarily many times in a single execution. Initialization, killing, resumption, and termination of task S on core_1 is modeled by performing actions `iS1` (edge 1), `kS1` (edges 3–4), `rS1` (edge 2), and `tS1` (edge 5), respectively. Every task automaton has at least two locations: `Idle` and `Active`. The task is either killed or yet to be initialized in location `Idle`. Every non-idle location has an invariant to force the task to terminate within the WCET; for instance, an automaton modeling task S has invariant $x \leq wS$ to force termination, where global clock x records the amount of time passed since the last initialization of S and global constant wS stores the WCET of S. Similarly, global constant bS stores the BCET of task S. Hence, clock guard $x \geq bS$ prevents task S to terminate before the BCET (edge 5).

Task Specific Activities (edges 6–8) Task S records exactly one update of a digital speedometer (modeled as global variable vS) in an execution: vS represents the speed in multiples of five varying from zero to hundred. Boolean variable uS is 1 if and only if the speedometer is updated in the current execution of S.

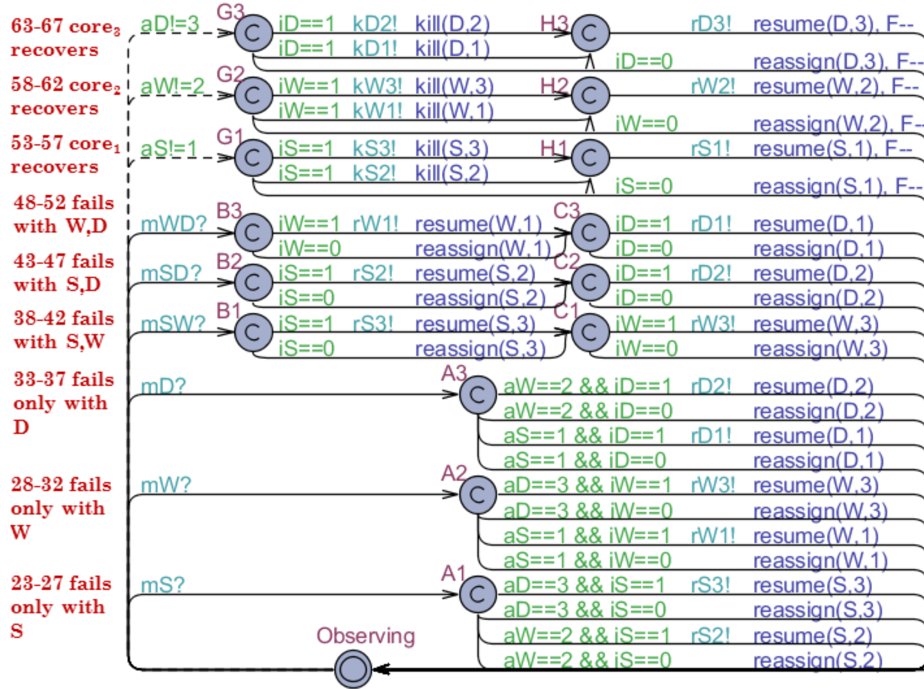
Task automata $core_1$.W and $core_1$.D in the concrete model is presented in the appendix. The automata model task life-cycle activities and task specific activities of tasks W and D.

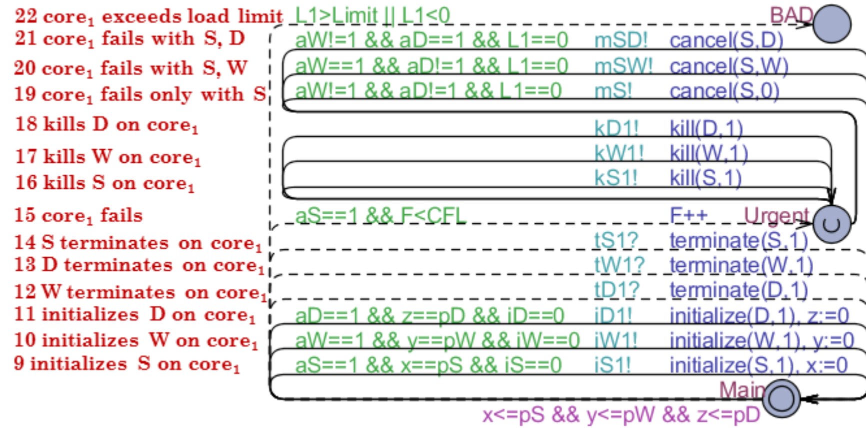
Core Automata A core automaton in the concrete model models two types of activities:

Operation-Time Activities (edges 9–14) A core automaton periodically initializes a task at its release period if the corresponding core is assigned to execute that task. Task terminates voluntarily after completing its assigned functions. A task between its initialization and termination occupies a portion of the resources. When a task terminates (resp., is initialized) on a core, the corresponding core automaton decreases (resp., increases) a variable modeling the current worst-case load. In location Main, task S is initialized by performing action iS1 (edges 9) if S is assigned to $core_1$ (aS==1), and S is not initialized yet (iS==0), and clock x hits the value of the release period of S (x==pS). Automaton $core_1$ (edge 14) receives action tS1 from task automaton $core_1$.S (edge 5) whenever S terminates its execution on this core. Function terminate(S,1) decreases (resp., initialize(S,1) increases) variable L1, modeling the worst-case load on $core_1$, by constant lS1, the worst-case load of task S on core $core_1$, and resets Boolean variable iS to 0 (resp., 1), that means task S terminates (resp., is initialized).

Failure-Time Activities (edges 15–22) Core automaton $core_1$ models failures of the core by traversing an uncontrollable edge. In order to reflect our assumption on the concurrent failure limit, this edge is only admitted if the number of currently failed cores (F) is less than CFL (CFL). Location Urgent is reached from Main whenever $core_1$ fails. Urgent is one of the urgent locations[4], denoted as Ⓤ in Uppaal Tiga syntax, that means the automaton cannot spend time in this location (edges 15–21). When the core fails, the automaton instantaneously kills all tasks currently released by it—to simulate that no task can continue to run on a failed core (edges 16–18). Then the automaton instantaneously performs specific actions to broadcast a message containing the list of currently assigned tasks to that failed core: performs action mS if S is the only assigned task; action mSW if S and W are the only assigned tasks; and action mSD if S and D are the only assigned tasks (edges 19–21). To note that only task W or task D or both cannot be assigned to core $core_1$ without task S because a task (S) must be assigned to its operational primary core ($core_1$). At runtime, the reconfiguration services use a distributed monitoring system to identify these (task) assignments because no failed core can broadcast a message. An unsafe location BAD becomes

[4] Semantically, urgent locations are equivalent to: adding an extra clock, x, that is reset $x := 0$ on every incoming edge, and adding an invariant $x <= 0$ to the location.





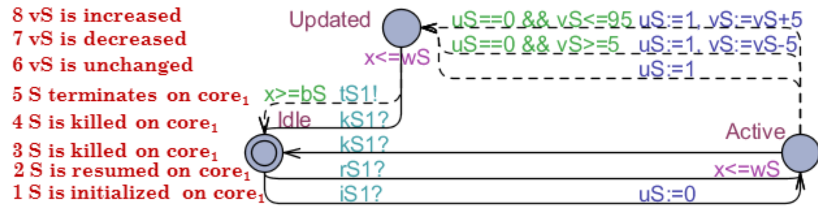


Fig. 5. Automata $\mathsf{core}_1.\mathsf{S}$ (in the bottom), core_1 (in the middle), `service` (in the top)

reachable when the current worst-case load on core_1 exceeds the load limit of

core_1 because of the failure of some other core(s) (edge 22). This prevents the synthesis algorithm from producing a strategy that would require illegal loads.

Service Automaton A `service` automaton spends most of its time in observing states waiting for a fault to occur (or for a core recovery from a temporary failure). The automaton reallocates a task in two steps: (i) assigns the periodic execution of the task to a suitable operational core, and (ii) resumes the task on the assigned core if the task was initialized before the reallocation. Other than `Observing` all locations are committed[5], denoted as Ⓒ in Uppaal Tiga syntax. They model states when reconfiguration decisions are taken, which are expected to be instantaneous and get precedence even over the urgent transitions of the other automata. Activities of the automaton can be divided into three groups described in the following.

Handling a Primary Core Failure (edges 23–37) Recall the invariant that an operating core is always assigned to execute its primary tasks, so in system_1 when a core (say core_1) is assigned to execute only one task then it must be a primary task (`S`). In the model a failure message is broadcast using an action (e.g., `mS`, `mSW`, and `mWD`) linked to the currently assigned tasks of the failed core, instead of the name of the core. Therefore, whenever a core failing with only assignment of the periodic execution of task `S` (or action `mS` is performed) then core_1, the primary core of `S`, must be that failed core. At that point, task `S` is reallocated to either core_2 or core_3. For example, location `A1` is reached from location `Observing` when core_1 fails (edges 23–27); in `A1` the target is reallocating `S`, the primary task of core_1, to core core_2 (bottom two outgoing edges) or to core core_3 (top two outgoing edges). Details of reallocation depending on whether the task was initialized (and needs to be reassigned then resumed) or was yet to be initialized (and just needs to be reassigned). For instance, to reallocate task `S` to core_2, location `Observing` is reentered from `A1` by: (i) assigning the periodic execution of `S` to core_2 (`aS:=2`) if core_2 is operational (`aW==2`) and `S` was yet to be initialized (`iS==0`), or (ii) assigning the periodic execution of `S` to core_2 and resuming `S` on the core (by performing `rS2`) if core_2 is operational and `S` was initialized (`iS==1`).

Handling a Backup Core Failure (edges 38–52) In our example, when a core is assigned to execute two critical tasks then one of them must be a backup task of that core; hence, after such a failure at least two cores concurrently are in their failed states. The fault model does not allow all cores to fail concurrently. For instance, core_1 must be operational when core_2 and core_3 are in their failed states; and the executions of tasks `W` and `D` have to be assigned to core_1. Location `B1` is reached from `Observing` when a core fails that is responsible to execute both `W` and `D` or when action `mWD` is received (edges 48–52). Location `C1` is reached from

[5] A commited location is the same as a urgent location but after reaching a committed location the next transition must involve an outgoing edge of at least one of the committed locations of the network.

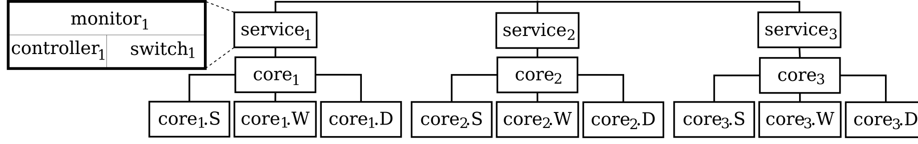


Fig. 6. Architecture of system_1 at runtime

`B1` by assigning the periodic execution of `W` to the only operational core core_1 and resuming `W`, if necessary (`iW`==1). Then `Observing` is reached by assigning the periodic execution of `D` to core_1 and resuming `D`, if necessary (`iD`==1).

Handling a Primary Core Recovery (edges 53–67) The periodic execution of a task must be assigned to its primary core when it is operational. Therefore, a task must be reallocated from a backup core to the primary core whenever it recovers from a temporary failure. The periodic execution of task `S` can be assigned to a backup core (`aS`!=1) only if its primary core core_1 is in a failed state. Location `G1` is reached from location `Observing` when core_1 recovers from a failure (edges 53–57). In `G1` the controller has two main choices depending on the initialization condition of the task: `S` is yet to be initialized and needs to be only reassigned to its primary core (the bottom outgoing edge); and `S` is initialized on a backup core and needs to be killed (the top two outgoing edges) then to be resumed on the primary core (the outgoing edge from location `H1`).

2.4 Synthesis

We synthesize reconfiguration services in three sequential steps: 1) generate a central controller for critical tasks, 2) construct a distributed controller for each core by exclusively distributing the central controller, and 3) synthesize a reconfiguration service for each core by adding its distributed controller with a constructed monitor to broadcast its health messages and a constructed switch to suspend and reinstate its non-critical tasks. A reconfiguration service runs on a core, which can fail. Hence, fault tolerance cannot be achieved using only one central reconfiguration service. We propose for each core to execute its own reconfiguration service that has three components: a distributed controller to reallocate critical tasks, a monitoring system to observe the system's conditions, and a edge to cancel and reinstate the periodic execution of non-critical tasks. All the distributed controllers of a system differ from each other—but complement each other in a way that they all together work similarly with a central controller, which is synthesized by analyzing the timed game model of Section 2.3. Figure 6 presents the architecture of system_1 with the reconfiguration services at runtime.

Central Controller Synthesis We perform a controller synthesis for the monolithic model of Section 2.3 against a safety objective which states that there is a strategy to always avoid locations core_1`.BAD`, core_2`.BAD`, and core_3`.BAD`. If

the property holds, the strategy—which is our central controller—is automatically synthesized by a timed game solver.

In order to obtain the most fault-tolerant controller possible, we synthesize it for the maximal concurrent-failures–limit (MCFL), the maximal value of CFL for which such a controller still exists. We use binary search to find MCFL. If MCFL is zero, no safe controller exists. The higher MCFL implies the better fault-tolerance by the reconfiguration services. The value of MCFL is strictly bounded by the total number of processing cores. Consider, for instance, configuration C1 in Table 1[6] where the release period, the WCET, the BCET of every task is 10, 5, and 4 time units, respectively; the worst-case load of tasks S, W, and D on core_1 (resp., core_2, core_3) are 60% (resp., 10%, 10%), 45% (resp., 80%, 5%), and 5% (resp., 5%, 85%), respectively. Configuration C1 does not have a controller for CFL 2. However, there is a controller for CFL 1. Maximal concurrent-failures–limit for system_1 for configuration C1 is 1 because 1 is the maximal value of CFL for which a controller exists.

Service Synthesis We synthesize the distributed reconfiguration service of a core by combining its distributed controller with an embedded monitor and an embedded switch.

Distributed Controller The functions of the central controller are completely and exclusively distributed into separate controllers for each core. A distributed controller is responsible for killing, reassignment, and resumption of critical tasks only on its core. A timed game represents all the possible transitions of the controller. As a result, a timed game may have non-deterministic choices for the controller. For example, in Figure 3 the controller has non-deterministic choices at system-state s_4 when only core_2 fails and the other two cores are operational (edges 28–32). A strategy removes non-determinism for the controller. By directing the controller to take the correct paths, a strategy plays a crucial role when in the model some paths guarantee satisfaction of a property (say reallocating task W to core_3 at system-state s_5 in Figure 3) and some paths do not (say reallocating W to core_1). For example, when core_2 fails (edges 28–32) a strategy (or the central controller) may say, "`if the system-state fulfills condition` X `then reallocate task W to` core_3`, otherwise to` core_1"; then the distributed controller of this portion (edges 28–32) for core_3 is "`if the system-state fulfills condition` X `then reallocate task W to` core_3"; and the distributed controller of this portion (edges 28–32) for core_1 is "`if the system-state does not fulfill condition` X `then reallocate task W to` core_1". Thus, deriving the distributed controllers from the central controller is a mechanical process and cannot fail.

Monitor The monitor of a reconfiguration service periodically broadcasts health messages of the corresponding core. A health message has three parts: (a) name

[6] To show clearer impacts of different modeling aspects on the analysis, we picked some imaginary system configurations instead of some actual system configurations.

of its core, (b) currently assigned critical tasks to its core, and (c) currently initialized critical tasks on its core. A monitor periodically also receives health messages—from the other reconfiguration services—and manipulates received messages. It marks a core as a failed core if two consecutive health messages of that core are not received. The monitor identifies a core recovery when it receives a message from a previously failed core. In the same way, the monitor detects when the scheduler releases a task and when a task terminates on a core.

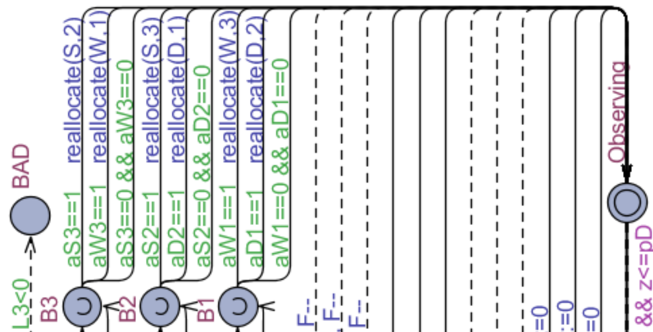
Switch A reconfiguration service has a static lookup table and a dynamic lookup table. The static lookup table lists the worst-case core load of every critical task (of the system) on this core and of every non-critical task assigned to this core. The dynamic lookup table keeps updated list of the assigned tasks, temporarily suspended non-critical tasks, and permanently suspended non-critical tasks. The controllers reallocate critical tasks from a failed or to a recovered core without considering the existence of non-critical tasks. The switch of a reconfiguration service (of the targeted core) suspends a set of non-critical tasks on its core using the lookup tables when the residual capacity on the core is insufficient to run the newly reallocated task safely. The distributed controllers first take necessary steps related to primary tasks of the recovered core whenever a core recovers. After that the switches reinstate the periodic executions of a set of suspended non-critical tasks on each source core where free processing capacity is revived due to the recovery. The switch permanently suspends a non-critical task when it breaches safety constraints.

2.5 Manual State-Space Reduction

The scalability of our service synthesis process mostly depends on the central controller synthesis as the remaining steps are mechanical and cannot fail. The concrete model has very large state space. For example, configuration C1 in Table 1 generates a strategy of size 290,663 KB in 94.20 seconds for this model when CFL is 1, presented in Table 2. Moreover, for many configurations the solver runs out of memory during analysis, such as, C3–C5 in Table 2. Detailed and monolithic models like the concrete model are easy to construct, understand, and present. However, large state spaces make them a poor choice for analysis.

The main purpose of the strategy is to resolve non-determinism among enabled controllable transitions in a way that guarantees satisfaction of the desired property. Hence, one can abstract away every detail from a timed game model that does not contribute to the non-determinism (or to the property). For instance, task specific activities and their non-deterministic updates of the tasks, which do not have any impact on our property, can be removed from a timed game model of system_1. Using such aggressive abstractions we construct the abstract model of system_1. Presented in Figure 7, the model has only one automaton.

The abstract model uses all the modeling



abstractions and system parameters of Section 2.3. Explicitly it models only task initializations (edges 68–70), task terminations (edges 71–76), core failures (edges 77–79), core recovery (edges 80–94), and safety violations (edge 95). Like task killings and resumptions, task initializations and terminations change the load on a core; thus they play an important role in the required property (or the safety checking). The invariant is used to release or initialize the tasks periodically. While a task termination within the WCET is forced by allowing an additional controllable transition (edges 74–76). Reallocation is a function which combines task killings, reassignments, and resumptions (edges 77–94). The model uses nine Boolean variables aS1, aW1, aD1, aS2, aW2, aD2, aS3, aW3, and aD3 to keep track the currently assigned tasks to cores: the value of aS1 (resp., aW1, aD1) is 1 when the periodic execution of task S (resp., W, D) is assigned to core core_1, otherwise the value is 0; similarly, aS2 (resp., aW2, aD2) is 1 if and only if the periodic execution of task S (resp., W, D) is assigned to core core_2. If both the concrete model and the abstract model use a variable or constant, it is used for the same purpose; for example, variable iS in both the models is used to identify when task S is initialized.

For the control problem described in this section, we constructed four different models: the concrete model as described in Section 2.3, the abstract model as described in this section, the monolithic model, and the compositional model.

Configuration	Period of task			WCET of task			BCET of task			Load on core_1 of task			Load on core_2 of task			Load on core_3 of task		
	S	**W**	**D**	**S**	**W**	**D**	**S**	**W**	**D**	**S**	**W**	**D**	**S**	**W**	**D**	**S**	**W**	**D**
C1	10	10	10	5	5	5	4	4	4	60	45	5	10	80	5	10	5	85
C2	10	10	10	5	5	5	0	0	0	60	45	5	10	80	5	10	5	85
C3	10	15	20	5	5	5	0	0	0	60	45	5	10	80	5	10	5	85
C4	10	15	20	5	5	5	0	0	0	60	35	5	10	80	5	10	5	85
C5	10	15	20	5	5	5	0	0	0	43	37	7	11	67	19	23	13	59
C6	10	15	20	5	5	5	0	0	0	43	37	59	11	67	39	23	13	59
C7	10	15	20	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C8	10	15	30	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C9	10	20	30	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C10	11	19	31	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C11	5	7	11	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C12	5	7	11	5	3	2	0	0	0	33	33	33	33	33	33	33	33	33
C13	5	7	11	5	3	2	5	3	2	33	33	33	33	33	33	33	33	33
C14	10	15	20	5	5	5	5	5	5	33	33	33	33	33	33	33	33	33
C15	10	15	20	5	7	11	5	7	11	33	33	33	33	33	33	33	33	33
C16	10	15	20	5	7	11	0	0	0	33	33	33	33	33	33	33	33	33
C17	10	15	20	7	7	7	7	7	7	33	33	33	33	33	33	33	33	33
C18	10	15	20	5	7	7	5	7	7	33	33	33	33	33	33	33	33	33
C19	10	15	20	7	7	11	7	7	11	33	33	33	33	33	33	33	33	33
C20	10	15	20	9	13	19	9	13	19	33	33	33	33	33	33	33	33	33

Table 1. Different configurations: combinations of release period, WCET, and BCET have abstract time units; and loads are in % of the respective core

The last two models are presented in Section 3. We analyze these models with many configurations. This section discusses behaviors of the concrete and abstract models for 20 configurations of Table 1.

All the analyses and controller syntheses were performed by Uppaal Tiga-0.17 on a PC with an Intel Core i3 CPU at 2.4 GHz, 4 GB of RAM, and running 64-bit Windows 7. We compare the concrete and abstract models with respect to controller synthesis time and the strategy size. Uppaal Tiga(-0.17) generates the same (size of) strategy for the same configuration on the same machine. Controller synthesis time, on the contrary, varies a little for the same configuration on the same machine. Therefore, we synthesize a strategy for every configuration multiple times, and then take the average synthesis time for each configuration.

Experimental results of the concrete and abstract models are presented in Table 2. We have the following six observa-

Config. of Table 2	CFL	Comparison			
		concrete model		abstract model	
		time	size	time	size
C1	2	No controller exists			
	1	94.20	290663	0.08	73
C2	2	No controller exists			
	1	115.71	296524	0.11	107
C3	2	No controller exists			
	1	Out of memory		0.14	242
C4	2	Out of memory		0.25	712
	1			0.14	266
C5	2	Out of memory		0.25	712
	1			0.14	266

tions from this table: **OB1A)** The abstract model improves the scalability dramatically for every configuration of Table 1. Other than aggressive abstraction, it encodes the whole model into only one automaton to avoid parallel composition, because parallel composition typically increases the size of the state space rapidly. **OB2A)** The larger the difference between WCET and BECT the longer the analysis time, and the sparser the strategy. Consider, for example, configuration C1 versus configuration C2, C7 versus C14, C12 versus C13, and C15 versus C16. **OB3A)** The smaller the least common multiples of release periods the smaller state space, the shorter analysis time, and the more compact strategy. Consider, for example, C2 versus C3, C8 versus C9, C9 versus C10, C10 versus C11, and so forth. For configurations C10 and C11, we use three different prime numbers as release times to get large least common multiples of the release times. As a result, for these configurations, we have sparse strategies along with long synthesis times even for the abstract model. One should check the least common multiples of the release times of a system before trying to (model and) synthesize controller for it using timed games. Unfortunately, timed games-based analytical tools are currently not mature enough to synthesize scheduler for practical systems having large least common multiples of the release times. **OB4A)** On the other

hand, the least common multiples of the execution times have no visible impact on the analysis time or the size of the strategy; (for instance, C14 versus C15, C15 versus C17, C17 versus C18, C18 versus C19, C19 versus C20, and so forth). **OB5A)** Variations in the least common denominator of non-clock variables, such as different loads, do not have any significant impact on the analysis; (for example, C4 versus C5 and C5 versus C7). **OB6A)** Uppaal Tiga takes less time and generates a smaller strategy for a higher value for CFL; (for instance, configurations C4, C5, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, and C20.)

Probably, the first observation **OB1A** is the most important one, which states that the scalability improves in the abstract model. Table 3 in Section 3 shows that the above observations are also true for the monolithic model and the compositional model. The MCFL of system system_1 depends on its configuration and model. For the concrete model, the MCFL is unknown for configurations C3, C4, C5, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, and C20; 1 for configurations C1 and C2; and 0 for configurations C6. For the abstract model the MCFL is 2 for configurations C4, C5, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, and C20; 1 for configurations C1, C2, and C3; and 0 for configurations C6.

2.6 Discussion

We briefly discuss the handling of systems with slightly different properties. For systems with asymmetric cores, which are unable to execute some tasks on some of the cores, we simply do not model the initialization, termination, killing, reassignment, and resumption for the illegal combinations of tasks and cores. For symmetric multi-core processing (SMP) one simply has to set the same load parameters on all the cores for each task. The synthesized reconfiguration services are oblivious to the tasks having substructure (sub-tasks), if they can be consistently abstracted by a single set of parameters (WCET, BCET and load).

We have assumed that an initialized task reallocated from a failed core should resume in the same state. If this is not required, i.e., a task can start from initial state on the new core at its next release period, then the model can be simplified, by removing the edges modeling resumption. We have not investigated the synthesis process for a scheduler with a dynamic allocation yet. In the next section, we present a theoretical framework for dynamic hierarchical open systems (such as system system_1) having any numbers of control hierarchies. The framework supports an automated state space reduction technique to allow timed games-based analysis for industrial dynamic hierarchical open systems.

3 Timed Process Automata

This section develops a model for the *automated analysis* of *safety* and *reachability* properties in industrial *time-critical systems*. To fulfill industrial requirements, we consider time-critical systems that are open (communicate with external

components), hierarchical (can be decomposed and recomposed into smaller control systems), and dynamic (the decomposition can change over time). In the section, we use *real-time systems*, meaning time-critical systems that fulfill all these features. The model also facilitates compositional modeling with reuse for different contexts.

Modeling techniques, automated analyses, and other key issues of TA are typically addressed for *static closed systems*. The application domain of TA is growing [19]. In our two projects [31,6] with an automotive manufacturer, we used different TA-based analyses to investigate the fault-tolerance of real-time systems, which are part of many large-scale safety-critical systems. During our industrial projects, we observed that continuous-time formal methods of TA may provide the most accurate analysis; however, TA are not suited for industrial real-time systems mainly because of poor *scalability*. Moreover, we found that TA may introduce cumbersome design details in a large-scale real-time system having several control hierarchies. This section extends TA to achieve better modeling support and scalability for automated analysis of hierarchical open real-time systems.

We propose *timed process automata (TPA)*, a variant of TA, for the development of industrial hierarchical open real-time systems [32]. The proposed variant provides compositional modeling with reuse for three different contexts and automatable analysis—a system needs to be modeled and analyzed using TPA only once when copies of it are used as independent systems or multiple components of a larger system or components of different larger systems or a combination of all previous scenarios.

3.1 Processes

Timed process automata model processes in a way that each process is a dynamic hierarchical open time-critical system, which we simply call real-time system in this section. Every process hierarchically contains its active callee processes. Thus the control of a process is hierarchically shared with its active callee processes. The main thread of a process can activate callee processes via communication channels. An active process can receive any input in any state. An active callee process can deactivate itself in any state of the main thread of its caller process. An activated callee process terminates within its worst-case execution time. This section presents the syntax and the semantics of TPA.

Timed Process Automata Timed process automata are a variant of TIOA. Unlike a TIOA, a timed process automaton has a finite set of *start actions* A_{s}, a finite set of *finish actions* A_{f}, a final location l_{f}, and a finite set of *channels* $\boldsymbol{C}$.

The set of *actions* $A = A_{\mathrm{i}} \oplus A_{\mathrm{o}} \oplus A_{\mathrm{s}} \oplus A_{\mathrm{f}}$ of a timed process automaton is a disjoint union of finite sets of input actions A_{i}, output actions A_{o}, start actions A_{s}, and finish actions A_{f}. For every set of actions A, there exists a bijective mapping between its start actions A_{s} and finish actions A_{f} in such a way that for each start action $\mathrm{s}_N \in A_{\mathrm{s}}$ there is exactly one finish action $\mathrm{f}_N \in A_{\mathrm{f}}$, and vice versa. These

actions can be used for starting and finishing processes associated with N. We use s and f with the name N (of another timed process automaton) as a subscript index (e.g., s_N and f_N) to denote a start action and a finish action, respectively. We use the same subscript to indicate *paired* actions. We write a to denote an action in general. Processes synchronize via instantaneous channels. Each timed process automaton uses the same designated symbols for its *public channel* ($*$) and *caller channel* ($\triangle$). We use c to denote a channel in general.

Definition 5 *A* timed process automaton *is a tuple* $T = (L, l_0, X, A, C, E, I, l_\mathrm{f})$, *where* L *is a finite set of locations,* $l_0 \in L$ *is the initial location,* X *is a finite set of clocks,* $A = A_\mathrm{i} \oplus A_\mathrm{o} \oplus A_\mathrm{s} \oplus A_\mathrm{f}$ *is a finite set of actions as described above,* C *is a finite set of channels,* $E \subseteq (L \times A \times C \setminus \{\triangle, *\} \times \Phi(X) \times 2^X \times L) \cup (L \times (A_\mathrm{i} \cup A_\mathrm{o}) \times \{\triangle, *\} \times \Phi(X) \times 2^X \times L)$ *is a set of edges,* $I : L \to \Phi(X)$ *is a total mapping from locations to invariants, and* $l_\mathrm{f} \in L$ *is a designated final location which does not have any outgoing edges to other locations and has the invariant* $I(l_\mathrm{f}) = \mathit{true}$.

Figure 2 presents TPA Brake-by-Wire, Position, and Actuator. In the figure, each initial location has a dangling incoming edge, final locations are filled with black, and TPA names are underlined. The final location l_f of a timed process automaton may be unreachable from the initial location (and then l_f is not shown in the figure).

Process Executions Every instance of a timed process automaton is a *process*. Two processes of the same timed process automaton represent two different copies of the same system. Every process has a unique *process identifier*. A *process* is a tuple $P = (\mathsf{id}(P), \mathsf{tpa}(P), \mathsf{channel}(P))$, where $\mathsf{id}(P)$[7] is the process identifier, timed process automaton $\mathsf{tpa}(P)$ defines the execution logic, and *caller channel* $\mathsf{channel}(P)$ is the private channel to communicate with the caller and the other processes which are started via the same channel. A process Q is a *callee* of P if P is the caller of Q. We use $\perp$ to denote the caller channel of the root process. Every process P of $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_\mathrm{f})$ has its own copy $P.c$ of channel $c \in C$. We write $P.c.a$ meaning that action a is performed via channel $P.c$.

At the same time, no two processes of the same timed process automaton can have the same caller channel. A process P, therefore, may have at most $|C| \times |A_\mathrm{s}|$ active callee processes. For example, an instance of automaton Brake-by-Wire of Figure 2 can activate at most two instances of automaton Position of Figure 2 at the same time via two different channels front and rear, where the instance of Brake-by-Wire is the caller process of the two instances of Position, which are the callee processes of the instance of Brake-by-Wire. A *subprocess* is a callee or a callee of a subprocess, recursively. For example, every instance of Brake-by-Wire has six subprocesses: two instances of Position and four instances of automaton Actuator of Figure 2. Every process hierarchically contains all of its subprocesses. Two processes are *siblings* if they have the same caller channel. The caller can use separate channels to differentiate control over different callees, even if they are processes of the same automaton.

[7] To avoid clutter, we abuse notation by writing P instead of $\mathsf{id}(P)$.

A process P starts a process Q of an automaton $\mathsf{tpa}(Q)$ via channel $P.c$ by traversing an edge $e_1 = (_, s_{\mathsf{tpa}(Q)}, c, _, _, _)$ labeled by a start action $s_{\mathsf{tpa}(Q)}$ if there exists no active process of tpa(Q) with caller channel $P.c$; dually, P traverses an edge $e_2 = (_, f_{\mathsf{tpa}(Q)}, c, _, _, _)$ labeled by the paired finish action $f_{\mathsf{tpa}(Q)}$ whenever Q reaches its final state. No edge labeled by $f_{\mathsf{tpa}(Q)}$ will ever be traversed if $\mathsf{tpa}(Q)$ is a *non-terminating timed process automaton*. Correspondingly, note that existing processes may start different processes of $\mathsf{tpa}(Q)$—but always with different process identifiers. However, only P listens to finish action $f_{\mathsf{tpa}(Q)}$ via channel $\mathsf{channel}(Q)$. Process P traverses an edge $e = (_, a, c, _, _, _)$ when P receives (respectively, sends) an input (resp., output) a in channel $P.c$. Process P communicates with its callee Q via $\mathsf{channel}(Q)$ and with the environment via channel $P.*$.

We formalize the above mechanics of execution by first giving the semantics of the main thread of the process, ignoring its subprocesses in Definition 6 and then giving the semantics of the entire process in Definition 7. The standalone semantics of a process are essentially the same semantics as a standard TIOA [3,12,13,9]. The main difference is that states are decorated with process identifiers and edges with channel names to distinguish different instances of the same timed process automaton in Definition 7. Also the caller channel $\triangle$ is instantiated for an actual parent process. The technical reason for this will become apparent in Definition 7.

Definition 6 *The* standalone semantics $\mathcal{S}[\![P]\!]$ *of a process* $P = (P, \mathsf{tpa}(P), \mathsf{channel}(P))$ *is a timed I/O transition system* $\mathcal{S}[\![P]\!] = (L \times \mathbb{R}^X_{\geq 0} \times P, (l_0, \mathbf{0}, P), A^P, \twoheadrightarrow)$[8]*, where* $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_f)$*,* $\mathbf{0}$ *is a function mapping every clock to zero and* $\twoheadrightarrow \subseteq (L \times \mathbb{R}^X_{\geq 0} \times \{P\}) \times (A^P \cup \mathbb{R}_{\geq 0}) \times (L \times \mathbb{R}^X_{\geq 0} \times \{P\})$ *is the transition relation generated by the following rules:*

Action *For each clock valuation* $v \in \mathbb{R}^X_{\geq 0}$ *and each edge* $(l, a, c, \phi, \lambda, l') \in E$ *such that* $v \models \phi$*,* $v' = v[x \mapsto 0]_{x \in \lambda}$*, and* $v' \models I(l')$ *we have* $(l, v, P) \xrightarrow{P.c.a} (l', v', P)$ *if* $c \neq \triangle$*, otherwise* $(l, v, P) \xrightarrow{\mathsf{channel}(P).a} (l', v', P)$

Delay *For each clock valuation* $v \in \mathbb{R}^X_{\geq 0}$ *and for each delay* $d \in \mathbb{R}_{\geq 0}$ *such that* $(v + d) \models I(l)$ *we have* $(l, v, P) \xrightarrow{d} (l, v + d, P)$*.*

Theorem 1 *The transition system induced by the standalone semantics of a process is time deterministic, time reflexive, and time additive*

Proof. The Action transition rule does not allow hidden or internal transition. All non-action and delay related state changes, thus, occur according to the Delay transition rule. From this rule we can derive:

- The only state that can be reached from state (l, v, P) after delaying $d \in \mathbb{R}$ time units is $(l, v + d, P)$,

[8] A^P is the set of actions where action names are constructed using regular expression $(P$"."$C \mid \mathsf{channel}(P))$"."$A$.

– The only state that can be reached from state (l, v, P) after delaying 0 time unit is (l, v, P), and
– For any two delays $d_1 \in \mathbb{R}$ and $d_2 \in \mathbb{R}$, the only state that can be reached from state (l, v, P) after delaying d_1 and d_2 time units is $(l, v+d, P)$ when $d_1 + d_2 = d$.

Therefore, the transition system induced by the standalone semantics of a process is time deterministic, time reflexive, and time additive.

Ground TPA are TPA that cannot perform a start or finish action ($A_\mathrm{s} \cup A_\mathrm{f} = \emptyset$). Automaton Actuator in Figure 2, for instance, is a ground timed process automaton. *Compound TPA* are TPA that can perform a start or finish action ($A_\mathrm{s} \cup A_\mathrm{f} \neq \emptyset$). For example, Brake-by-Wire and Position in Figure 2 are compound TPA. A *well-formed channel* cannot be used by two processes sharing an output action. Processes of a *well-formed timed process automaton* have only well-formed channels. *Non-recursive TPA* are defined inductively using the following rules: 1) every ground timed process automaton is a non-recursive timed process automaton, and 2) a compound timed process automaton which performs only those start and finish actions whose subscripts are the names of some other existing non-recursive TPA is a non-recursive timed process automaton. All three automata in Figure 2, for example, are non-recursive TPA. A process of a non-recursive timed process automaton hierarchically contains only a finite number of subprocesses. The caller may activate an idle process, iteratively. Thus a process may activate a subprocess an arbitrary number of times. In this section, we are only concerned with non-recursive well-formed TPA.

A *standalone final state* of a process P is (l_f, v, P), where v is any clock valuation. We use st^P, st_0^P, c^P, and st_f^P to denote a standalone state, the standalone initial state, the set of channels, and a standalone final state of process P, respectively. We say that a process P is A'-enabled for a channel $P.c$ if for every reachable standalone state st^P we have $st^P \overset{P.c.a}{\twoheadrightarrow} st'^P$ for some standalone state st'^P for each action $a \in A'$. We require that each process P is A_i-enabled (input enabled) for all channels of P, and A_f-enabled (finish enabled) for all channels of P other than channels $P.\triangle$ and $P.*$ to reflect the phenomenon that inputs from the environment and the deaths of callees are independent events, beyond the control of a process. We present the semantics of a process in the following:

Definition 7 *The* global operational semantics $\mathcal{G}[\![P]\!]$ *(semantics* $[\![P]\!]$ *for short) of a process* $P = (P, \mathsf{tpa}(P), \bot)$ *are a timed I/O transition system* $\mathcal{G}[\![P]\!] = (2^s, s_0, \mathbb{P} \times \mathbb{C} \times \mathbb{A}, \rightarrow)$*, where* s *is the set of all the standalone states of all the processes in the universe,* $\mathsf{tpa}(P) = (L, l_0, X, A, E, I, l_\mathrm{f})$*,* $s_0 = \{st_0^P\}$ *is the initial state,* $\mathbb{P}$ *is the set of all the processes in the universe,* $\mathbb{C}$ *is the set of all the channels in the universe,* $\mathbb{A}$ *is the set of all the actions in the universe, and* $\rightarrow \subseteq 2^s \times (\mathbb{P} \times \mathbb{C} \times \mathbb{A} \cup \mathbb{R}_{\geq 0}) \times 2^s$ *is the transition relation generated by the following rules:*

$$\frac{st^Q \overset{Q.c.\mathrm{s}_T}{\twoheadrightarrow} st'^Q \text{ and } c \notin \{\triangle, *\} \qquad \{st^W \in s \mid \mathsf{channel}(W) = Q.c \text{ and } \mathsf{tpa}(W) = T\} = \emptyset \qquad st^Q \in s \qquad (R, T, Q.c) \text{ is a freshly started process}}{s \xrightarrow{Q.c.\mathrm{s}_T} \{s \setminus \{st^Q\}\} \cup \{st_0^R, st'^Q\}} \textsc{Start}$$

$$\frac{\begin{array}{c} st^R_{\mathsf{f}}, st^Q \in s \text{ and } \mathsf{channel}(R) = Q.c \\ \{st^U \in s \mid \mathsf{channel}(U) \in C^R\} = \emptyset \qquad st^Q \xrightarrow{Q.c.\mathsf{f}_{\mathsf{tpa}(R)}} st'^Q \end{array}}{s \xrightarrow{Q.c.\mathsf{f}_{\mathsf{tpa}(R)}} \{s \setminus \{st^R_{\mathsf{f}}, st^Q\}\} \cup \{st'^Q\}} \textsc{Finish}$$

$$\frac{s' = \{st'^Q \mid st^Q \xrightarrow{d} st'^Q \text{ and } st^Q \in s \text{ and } (st^Q \neq st^Q_{\mathsf{f}} \text{ or } |s| = 1)\} \qquad |s| = |s'|}{s \xrightarrow{d} s'} \textsc{Delay}$$

$$\frac{a \notin \bigcup_{st^Q \in s} A_{\mathrm{o}}^{\mathsf{tpa}(Q)} \qquad s' = \{st^Q \in s \mid st^Q \xrightarrow{Q.*.a} st'^Q\}}{s \xrightarrow{a} \{s \setminus s'\} \cup \{st'^Q \mid st^Q \xrightarrow{Q.*.a} st'^Q \text{ and } st^Q \in s\}} \textsc{Input}$$

$$\frac{\begin{array}{c} st^Q \xrightarrow{W.c.a} st'^Q \text{ and } a \in A^Q_{\mathrm{o}} \text{ and } st^Q \in s \\ s' = \{st^R \in s \mid st^R \xrightarrow{W.c.a} st'^R \text{ and } W.c \text{ is a channel}\} \end{array}}{s \xrightarrow{Q.c.a} \{s \setminus s'\} \cup \{st'^R \mid st^R \xrightarrow{W.c.a} st'^R \text{ and } st^R \in s\}} \textsc{Output}$$

A *global state* is a set which holds standalone states of all active processes. The Start rule states that the initial standalone state of a freshly started callee is added to the global state whenever the corresponding start action is performed by its caller. The rule also states that no two active processes can have the same timed process automaton and the same caller channel. The Finish rule prescribes that the standalone-final state of a callee is removed from the global state and the caller executes the corresponding finish action whenever that callee is in the standalone-final state and no standalone state of its subprocesses is in global state. Thus the rule defines *global-final state* (*final state* for short) of a process: a process is in its the final state when the process is in its final location and the process has no active subprocess. The Delay rule declares that globally a process can delay if that process and all of its active subprocesses can delay in their respective standalone semantics. Every subprocess is a part of the root process and thus if a subprocess is performing an action (or not idle) then the root process is also not idle. The rule also says that a process cannot delay if that process or any of its subprocess is in its global final state. That means a process finishes as soon as it reaches its final state. The Input rule states that a process receives an input from the environment via channel $id.*$. Rule Output declares a process send an output via channel $id.c$ to others who share $id.c$.

Theorem 2 *The transition system induced by the global semantics is time deterministic, time reflexive, and time additive.*

Proof. The global semantics of a process of a ground timed process automaton is its standalone semantics. Therefore, the transition system induced by Definition 7 for that process is time deterministic, time reflexive, and time additive.

Standalone states of a subprocess can be part of global states only after that subprocess is started. Whenever a subprocess reaches its terminal state, its standalone states can never be part of the global state because of the Delay rule and the Finish rule. Therefore, a nonactive subprocess has no impact on the

transition system. None of the action transition rules allows hidden or internal transition. All non-action and delay related state changes, thus, occur according to the Delay transition rule. From this rule we can derive for a process P having n active subprocesses $P_1, P_2, \cdots, P_n$:

- The only state that can be reached from state $\{(l, v, P), (l_1, v, P_1), (l_2, v, P_2), \cdots, (l_n, v, P_n)\}$ after delaying $d \in \mathbb{R}$ time units is $\{(l, v+d, P), (l_1, v+d, P_1), (l_2, v+d, P_2), \cdots, (l_n, v+d, P_n)\}$,
- The only state that can be reached from state $\{(l, v, P), (l_1, v, P_1), (l_2, v, P_2), \cdots, (l_n, v, P_n)\}$ after delaying 0 time units is $\{(l, v, P), (l_1, v, P_1), (l_2, v, P_2), \cdots, (l_n, v, P_n)\}$, and
- For any two delays $d_1 \in \mathbb{R}$ and $d_2 \in \mathbb{R}$, the only state that can be reached from state $\{(l, v, P), (l_1, v, P_1), (l_2, v, P_2), \cdots, (l_n, v, P_n)\}$ after delaying d_1 and d_2 time units is is $\{(l, v+d, P), (l_1, v+d, P_1), (l_2, v+d, P_2), \cdots, (l_n, v+d, P_n)\}$.

Therefore, the transition system induced by Definition 7 is time deterministic, time reflexive, and time additive.

The process semantics, hence, defines a well-formed timed I/O transition system. This allows us to use TA as a basis for analyzing TPA.

A *local run* of the main thread of a process P is a standalone run of P for which there exists a global run of P such that every transition of that standalone run occurs in that global run. The *local behavior* of the main thread of P consists of all of its local runs.

3.2 Analysis

We are interested in *safety* and *reachability* properties of TPA. This section explains how such analyses can be performed using the theory of timed games. A standard TIOA can be viewed as a concurrent two-player timed game, in which the players decide both which action to play, and when to play it. The input player represents the environment, and the output player represents the system itself. Similarly, the main thread of a process acts as a concurrent two-player timed game: the environment plays input transitions and finish transitions, and the main thread of the process plays output transitions and start transitions. Let's consider interactions of a process defined in the previous section. A process controls its output and start transitions. After starting a callee, the main thread of the caller knows that the paired finish action will arrive within the worst-case execution time of the associated callee. However, the main thread does not have any control on the exact arrival time of a finish action. Finish transitions along with input transitions are uncontrollable. The environment of the main thread of a process consists of all the connected processes (such as caller, siblings, and subprocesses) and all unconnected entities.

A global state of a process is safe if and only if all of the standalone states which it holds contain no unsafe location. A *safety property* asserts that the system remains inside a set of global-safe states regardless of what the environment does. We are interested in *Safety Property I: Given a process P and a set of*

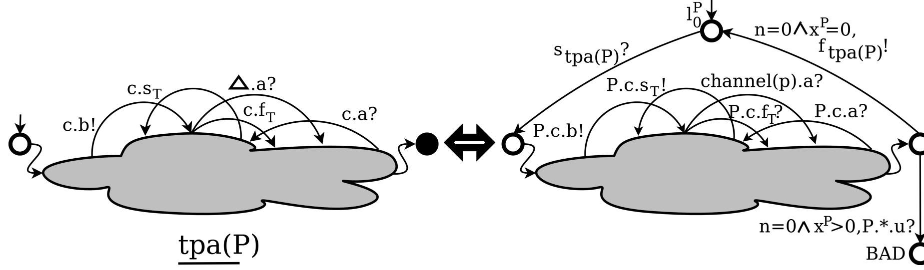


Fig. 8. A generalized view of the standalone automata construction

unsafe locations L_U of P, can the controller avoid L_U in P regardless of what the environment does?

A global state of a process is a target state if and only if at least one of its standalone states contains a target location. A *reachability property* asserts that the system reaches any of the global-target states regardless of what the environment does. We are interested in *Reachability Property I: Given a process P and a set of target locations L_T of P, can the controller reach a location of L_T in P regardless of what the environment does?*

The monolithic analysis constructs a static network of automata to represent all possible global executions by mimicking the hierarchical call tree of the analyzed process. It simulates a process execution by changing states of pre-allocated TIOA which fall into two groups: a *root automaton* to simulate the local behaviors of the main thread of the root process and a finite set of *standalone automata* to simulate the local behaviors of the main threads of the subprocesses.

Standalone Automata We construct a standalone automaton for each subprocess to simulate the main thread of that process. To construct a standalone automaton, we prefix the timed process automaton with a simulated start action and suffix it with a simulated finish action. We use non-negative finitely bounded integer variables[9] in standalone automata to count the number of active callees, in order to detect termination. We rename actions (e.g., a) of processes uniformly to encode channel names (e.g., $P.c$) in action names (e.g., $P.c.a$) of standalone automata; because standard TIOA do not support private channels. A standalone automaton includes all the locations and slightly altered edges of the corresponding timed process automaton. Moreover, each standalone automaton has two additional locations: a new initial location l_0^{id} to receive (resp., send) a start (resp., finish) message from (resp., to) the caller, and a new unsafe location BAD to prevent the automaton from waiting in final states instead of finishing. Every start (resp., finish) increments (resp., decrements) a counter variable n. The automaton represents finishing of the process in the final location when $n = 0$.

Definition 8 *The standalone automaton of process P is* $\mathsf{standalone}(P) = (L \cup \{l_0^{\mathrm{P}}, BAD\}, l_0^{\mathrm{P}}, X \cup \{x^{\mathrm{P}}\}, \{n\}, A^{\mathrm{P}}, E^{\mathrm{P}}, I^{\mathrm{P}})$*, where* $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_{\mathrm{f}})$*, l_0^{P} and*

[9] The use of non-negative finitely bounded integer variables can be avoided if a more cumbersome encoding is used.

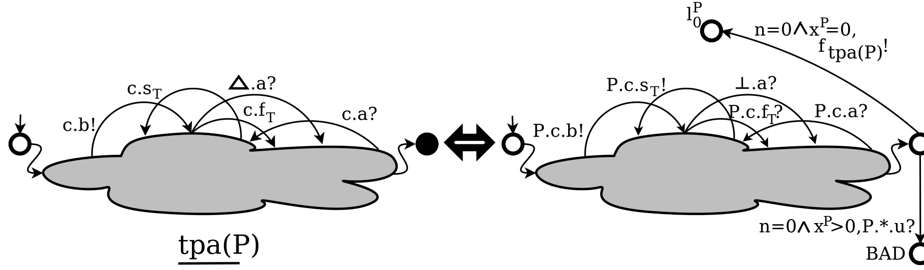


Fig. 9. A generalized view of the root automata construction

BAD are two newly added locations, x^{P} is a newly added clock, n is a non-negative finitely bounded integer variable with the initial value 0, $A_{\mathrm{o}}^{\mathrm{P}} = A'_{\mathrm{o}} \cup A'_{\mathrm{s}} \cup \{\mathsf{channel}(P).\mathrm{f}_{\mathsf{tpa}(P)}\}$ and $A_{\mathrm{i}}^{\mathrm{P}} = A'_{\mathrm{i}} \cup A'_{\mathrm{f}} \cup \{\mathsf{channel}(P).\,\mathrm{s}_{\mathsf{tpa}(P)}, P.*.u\}$ such that $A'_m = \{\mathsf{channel}(P).a \mid a \in A_m\} \cup \{P.c.a \mid a \in A_m \text{ and } c \in C \setminus \{\triangle\}\}$ where $m \in \{\mathrm{o}, \mathrm{s}, \mathrm{i}, \mathrm{f}\}$ and newly added actions are $\mathsf{channel}(P).\mathrm{s}_{\mathsf{tpa}(P)}$, $\mathsf{channel}(P).\mathrm{f}_{\mathsf{tpa}(P)}$, and $P..u$. The set of edges E^{P} contains*

- *Converted edges that do not communicate via caller channel $\triangle$:*
  - *An edge $(l, P.c.a, \phi, \xi, \lambda \cup \lambda', l') \in E^{\mathrm{P}}$ for each edge $(l, a, c, \phi, \lambda, l') \in E$, where $c \in C \setminus \{\triangle\}$, the integer assignment is empty $\xi = \emptyset$ when $a \in A_{\mathrm{o}} \cup A_{\mathrm{i}}$, $\xi = \{n - -\}$ when $a \in A_{\mathrm{f}}$, and $\xi = \{n + +\}$ when $a \in A_{\mathrm{s}}$*
- *Converted edges that communicate via caller channel $\triangle$:*
  - *An edge $(l, \mathsf{channel}(P).a, \phi, \emptyset, \lambda \cup \lambda', l') \in E^{\mathrm{P}}$ for each edge $(l, a, \triangle, \phi, \lambda, l') \in E$*
- *Additional new edges that simulate activation and deactivation:*
  - *Three more edges $(l_0^{\mathrm{P}}, \mathsf{channel}(P).\mathrm{s}_{\mathsf{tpa}(P)}, \emptyset, \emptyset, X, l_0)$, $(l_{\mathrm{f}}, \mathsf{channel}(P).\mathrm{f}_{\mathsf{tpa}(P)}, n = 0 \wedge x^P = 0, \emptyset, \emptyset, l_0^{\mathrm{P}})$, $(l_{\mathrm{f}}, P.*.u, n = 0 \wedge x^P > 0, \emptyset, \emptyset, BAD)$ are in E^{P}*

$\lambda' = \emptyset$ when $l' \neq l_{\mathrm{f}}$, otherwise $\lambda' = \{x^{\mathrm{P}}\}$. The invariant function I^{P} maps each location $l \in L$ to $I(l)$ and maps each location $l \in \{l_0^{\mathrm{P}}, BAD\}$ to true.

The standalone semantics of automaton $\mathsf{tpa}(P)$ and the semantics of standalone automaton $\mathsf{standalone}(P)$ are essentially the same in a way that both have the same safety and reachability properties (that we consider) of the corresponding process.

Root Automata We construct a root automaton to simulate the main thread of the analyzed process.

Definition 9 *To analyze a timed process automaton $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_{\mathrm{f}})$, we construct the root automaton $\mathsf{root}(\mathrm{P})$ of process P. Standalone automaton $\mathsf{standalone}(P)$ is slightly different from $\mathsf{root}(\mathrm{P})$. The differences are:*

- *The caller channel is always $\perp$,*
- *The initial location of root automaton $\mathsf{root}(\mathrm{P})$ is the location l_0, which is also the initial location of $\mathsf{tpa}(P)$, and*
- *Root automaton does not have edge $(l_0^{\mathrm{P}}, \perp.\mathrm{s}_{\mathsf{tpa}(P)}, \emptyset, \emptyset, X, l_0)$, which simulates activation of P.*

Monolithic Analysis Model Monolithic analysis models can be constructed in an automatable process.

Definition 10 *The monolithic analysis model of a ground timed processes automaton (such as* Actuator*) is its root automaton. We construct the monolithic analysis model of automaton* $\mathsf{tpa}(P)$ *in the following iterative manner:*

First Step: We construct the root automaton $\mathsf{root}(\mathrm{P})$.
Iterative Step: We construct a standalone automaton for each triple (Q, s_T, c), *where* Q *is process for which we have constructed a standalone automaton or the root automaton,* $\mathsf{tpa}(Q) = (L, l_0, X, A, C, E, I, l_\mathrm{f})$, $c \in C \setminus \{\triangle, *\}$, $\mathrm{s}_T \in A_\mathrm{s}$, *and* $(_, \mathrm{s}_T, c, _, _, _) \in E$.

Figures 8–9 present a generalized view of the standalone and root automata constructions (the appendix present monolithic analysis models of processes of automata Actuator, Position, and Brake-by-Wire). The monolithic analysis model constructs a parallel composition of all the TIOA constructed above. The construction is finite, and the composition is a TIOA, because we consider only non-recursive well-formed TPA.

We add avoiding *BAD* locations to our safety and reachability properties analyses. We convert Safety Property I to *Safety Property II: Given a process* P *and a set of unsafe locations* L_U *of* P*, can the controller avoid* L_U *and all the BAD locations in the analysis model regardless of what the environment does?* We also convert Reachability Property I to *Reachability Property II: Given a process* P *and a set of target locations* L_T *of* P*, can the controller reach a location of* L_T *in the analysis model avoiding all the BAD locations regardless of what the environment does?* Special actions are added with TPA to construct corresponding root and standalone automata to simulate starts and finishes of processes. Avoiding all the newly added *BAD* locations in the analysis model ensures that each caller process performs the corresponding finish action as soon as the callee finishes—exactly as described in the global semantics. Executions (of the analysis model) that avoid all the newly added *BAD* locations, when projected on the original alphabet, are identical to the executions of the global semantics. Thus, if a Safety Property I (resp., Reachability Property I) holds for a process then its corresponding Safety Property II (resp., Reachability Property II) also holds in the analysis model, and vice versa.

Definitions 8, 9, and 10 provide automatable techniques to construct standalone automata, root automata, and monolithic analysis models, respectively. Thus one can remove manual alterations—such as manual renaming—by making these constructions automatic.

3.3 Automatable State-Space Reduction

We introduce an automatable state-space reduction technique for TPA to counteract state-space explosion. The technique relies on compositional reasoning, aggressive abstractions, and reducing process synchronizations. In the monolithic analysis of Section 3.2, a callee can be represented by an arbitrary number of

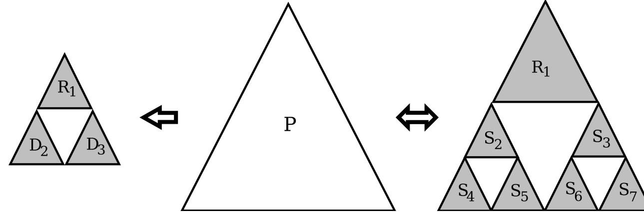


Fig. 10. A compositional (sound) analysis model on the left and a monolithic (sound and complete) analysis model on the right of automaton Brake-by-Wire, where P is a process of the automaton, R_1 is the root automaton, S_2–S_7 are standalone automata, and D_2–D_3 are duration automata

standalone automata, and each of these automata can be arbitrarily large. The compositional reasoning technique described in this section replaces hierarchical trees of standalone automata representing subprocesses with simple abstractions (Figure 10)—so called *duration automata*.

Duration Automata A duration automaton (Figure 11) is TIOA with only two locations: the initial location (l_0^P) and the active location (l_1^P). A duration automaton of an analyzed process abstracts all the information of global executions of the process other than its *worst-case execution time (WCET)*. It can capture safety and reachability properties of interest. The *minimal-time safe reachability* of a target location is the *minimal-time reachability* [33,34] for which the controller has a winning strategy to reach that target location by avoiding unsafe states. Like [35,36], we assume that the WCET $\mathbf{W}$ of a process P is the minimal-time safe reachability time to reach location l_0^{P} of automaton $\mathsf{root}(P)$ in the analysis model of P. The WCET of P is unknown ($\mathbf{W} = \infty$) when there is no winning strategy for the minimal-time safe reachability to reach location l_0^{P} of $\mathsf{root}(P)$.

Definition 11 *The duration automaton of process P is* $\mathsf{duration}(P) = (\{l_0^{\mathrm{P}}, l_1^{\mathrm{P}}\}, l_0^{\mathrm{P}}, \{x^{\mathrm{P}}\}, \emptyset, A^{\mathrm{P}}, E^{\mathrm{P}}, I^{\mathrm{P}})$, *where* $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_{\mathrm{f}})$, $A_{\mathrm{i}}^{\mathrm{P}} = \{\mathsf{channel}(P).\mathrm{s}_{\mathsf{tpa}(P)}\}$, $A_{\mathrm{o}}^{\mathrm{P}} = \{\mathsf{channel}(P).\mathrm{f}_{\mathsf{tpa}(P)}\}$, *the set of edges* $E^{\mathrm{P}} = \{(l_0^{\mathrm{P}}, \mathsf{channel}(P).\mathrm{s}_{\mathsf{tpa}(P)}, \emptyset, \emptyset, \{x^{\mathrm{P}}\}, l_1^{\mathrm{P}}), (l_1^{\mathrm{P}}, \mathsf{channel}(P).\mathrm{f}_{\mathsf{tpa}(P)}, \emptyset, \emptyset, \emptyset, l_0^{\mathrm{P}})\}$, *invariant* I^{P} *maps location* l_0^{P} *to true, and* I^{P} *maps location* l_1^{P} *to* $x^{\mathrm{P}} \leq \boldsymbol{W}$.

Compositional Analysis Model We construct the compositional analysis model in a bottom-up manner: analysis of a compound process is performed only after analyzing all its callees. Like the monolithic analysis, the compositional analysis model of a ground timed process automaton $\mathsf{tpa}(Q)$ (such as Actuator) is a

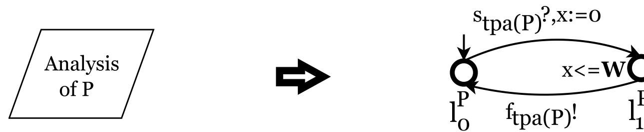


Fig. 11. A generalized view of duration automata construction

root automaton of process Q. That TIOA is analyzed to construct a duration automaton of Q. For a compound process P, we analyze automaton $\mathsf{root}(P)$ in the context of the duration automata of its callees (instead of the entire hierarchical structure of subprocesses).

Definition 12 *We construct the compositional analysis model of a timed process automaton* $\mathsf{tpa}(P)$ *in the following manner:*

First Step: We construct the root automaton $\mathsf{root}(\mathrm{P})$.
Second Step: We construct a duration automaton for each triple (P, s_T, c)*, where* $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_\mathrm{f})$*,* $c \in C \setminus \{\triangle, *\}$*,* $\mathrm{s}_T \in A_\mathrm{s}$*, and* $(_, \mathrm{s}_T, c, _, _, _) \in E$.

Figure 12 presents the compositional analysis procedure of Brake-by-Wire (the detailed models are presented in the appendix). The compositional model construction procedure terminates, and the composition of all the above TIOA is a TIOA, because we consider only non-recursive well-formed TPA.

Steps	Compositionl Analysis Models	Constructed Duration Automata
First	root(P0),tpa(P0)=Actuator	duration(P0)
Second	root(P1),tpa(P1)=Position || duration(P2),tpa(P2)=Actuator || duration(P3),tpa(P3)=Actuator	duration(P1)
Third	root(P4),tpa(P4)=Brake-by-Wire || duration(P5),tpa(P5)=Position || duration(P6),tpa(P6)=Position	

Fig. 12. Steps of the compositional analysis of automaton Brake-by-Wire: $\mathsf{root}(\mathrm{P}_0)$, $\mathsf{tpa}(\mathrm{P}_0) = \mathsf{Actuator}$ means root automaton of process P_0, where P_0 is an instance of Actuator, and similar interpretations apply for $\mathsf{root}(\mathrm{P}_1)$, $\mathsf{tpa}(\mathrm{P}_1) = \mathsf{Position}$, $\mathsf{duration}(\mathrm{P}_2)$, $\mathsf{tpa}(\mathrm{P}_2) = \mathsf{Actuator}$, and so forth.

The duration automaton of a process can capture safety properties: if a process has a winning strategy for a safety game, then both locations of its duration automaton are considered safe; otherwise, the active location (l_1^{id}) of the duration automaton is added to the set of unsafe locations L_U. Now this duration automaton can be used as a sound context to analyze the caller automaton for safety. A safety property holds for a compound process when the main thread of the process satisfies the property locally and allows the activation of a callee only if that callee also satisfies the property.

Duration automata can also capture reachability properties: if a process has a winning strategy for a reachability game then the active location (l_1^{id}) of the duration automaton is added to the set of target locations L_T; otherwise, no target location is specified for this callee. This duration automaton can be used as a sound context to analyze the caller automaton for reachability. A reachability

property holds for a compound process when the main thread of the process can reach the target locally or can activate a callee where the property holds. Like the monolithic analysis, the compositional analysis is performed for Safety Property II and Reachability Property II.

The compositional analysis is sound. A duration automaton does not contain any input and output actions of its process. Hence, the root automaton in a compositional model does not synchronize with the input and output actions of its callees—instead the automaton synchronizes for those actions with the environment. The duration automaton was created under the assumption that inputs are uncontrollable, so ignoring synchronization with inputs is sound. Similarly, it is sound to open the inputs of the root automaton from a callee, as they will be treated as uncontrollable and unpredictable actions, so will be analyzed in a more "hostile" environment than before the abstraction. Therefore, if a property holds in the compositional analysis then it also holds for the monolithic analysis. In other words, if a safety or reachability property holds in compositional analysis then it holds in the global semantics.

Our compositional analysis is not complete because it is based on potentially quite coarse abstractions. In compositional analysis, abstracting from the input and output actions of callees and subprocesses causes the process to be analyzed in a more "hostile" environment (i.e., an environment in which no assumptions whatsoever are made about the timing and relative order of these actions. Therefore, the process might have a winning strategy in its actual operating environment, when our compositional analysis produces the opposite result. Definitions 9, 11, and 12 provide automatable techniques to construct root automata, duration automata, and compositional analysis models, respectively. Thus one can automatically reduce state space by implementing our constructions.

3.4 Experimental Results

In all the steps of Figure 12, the largest composition contains only three automata, and except for the root automaton all are tiny duration automata. A monolithic analysis model of Brake-by-Wire is a composition of seven automata presented in Appendix B. A duration automaton always has a small constant size (modulo the size of the WCET constant), and so its state space is very simple (actually the discrete state space is independent of the input model).

First, we model the central reconfiguration service (in Figure 13) and three tasks: `S` (in the top of Figure 15), `W`, and `D` using TPA. The automaton in Figure 13 also models task releases (using start actions `sS`, `sW`, and `sD`) and terminations (using finish actions `fS`, `fW`, and `fD`). Like the concrete and abstract models of Section 2, an unsafe location `BAD` in this automaton is unreachable—a central reconfiguration service (or a controller) exists that makes the system fault tolerant—when the total load of no core can exceed its load limit. Similar to the concrete model, TPA of the tasks keep all the internal states of the corresponding tasks. Like the abstract model, the currently assigned core information is encoded into the automaton of Figure 13. These TPA together model system system_1 of Section 2

in a more abstract way than the concrete model but in a less abstract way than the abstract model.

After that, according to the construction technique of Section 3.2, we construct the standalone automata (presented in the middle of Figure 15) of the TPA representing tasks (presented in the top of Figure 15) and the root automaton (presented in Figure 13) of the timed process automaton representing the central reconfiguration service (presented in Figure 14). The composition of these four TIOA represents a monolithic analysis model of system system_1 of Section 2, and we simply call this model the monolithic model. Configurations of system system_1 of Section 2 are combinations of different worst-case loads of tasks on different cores, different worst-case execution times of tasks, different best-case execution times of tasks, and different release periods of tasks. Existence of a central reconfiguration service (or controller) depends on the current configuration. We analyze the monolithic model against 20 configurations of Table 2[10], which is a copy of Table 1. Like Section 2.5, all the analyses were performed by Uppaal Tiga-0.17 on a PC with an Intel Core i3 CPU at 2.4 GHz, 4 GB of RAM, and running 64-bit Windows 7. Table 3 represents the analysis results in the form of controller synthesis time (in seconds) and the strategy size (in kilobytes).

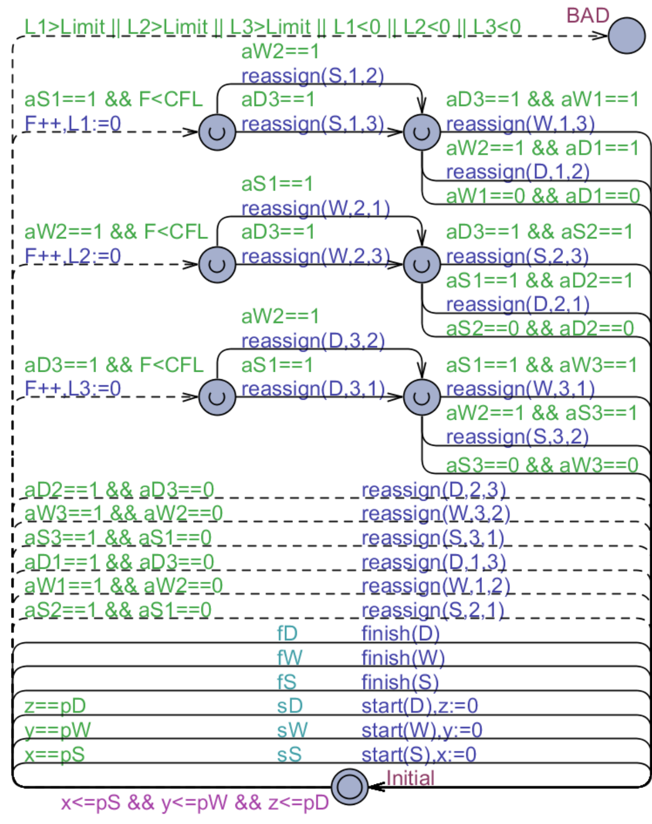


Fig. 13. A timed process automaton representing the central reconfiguration service

Unlike the previous section, we are not mainly concerned with controller synthesis from TPA—rather only checking the existence of a controller. We, however,

[10] To show clearer impacts of different modeling aspects on the analysis, we picked some imaginary system configurations instead of some actual system configurations.

also synthesized the controller because the synthesis time and the strategy size convey a clearer idea regarding the size of the state space. Moreover, they allow us to compare the models of this section with the models of the previous section. The monolithic model produces large state spaces, and for many configurations state-space explosion occurred, such as for configurations C3 (for CFL 1), C4, C5, C7, C8 (for CFL 2), C9, C10, C11, C12, C13, C14 (for CFL 2), C15 (for CFL 2), C16, C17 (for CFL 2), C18 (for CFL 2), and C19 (for CFL 2).

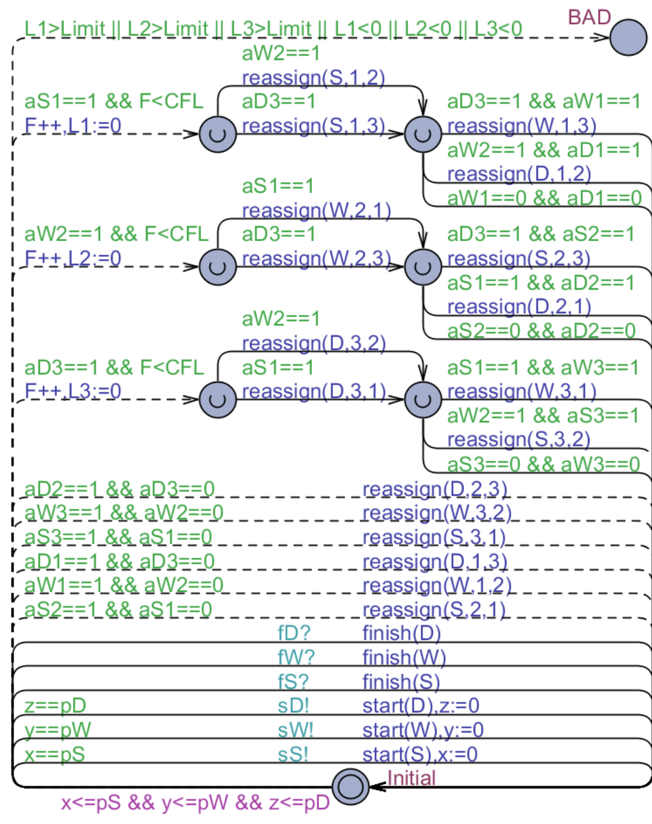


Fig. 14. Root automaton of the central reconfiguration service

At the end, according to the construction technique of Section 3.3, we construct the compositional model, which is a composition of the root automaton of the previous step and three duration automata (presented in the bottom of Figure 15). We performed the same experiments on the compositional model that we performed on the concrete model (in Section 2), the abstract model (in Section 2), and the monolithic model (in the previous step). Table 3 shows that the compositional model produces a much smaller state space than the monolithic model.

Experimental results of the monolithic and compositional models show: **OB1B)** Abstraction improves the scalability dramatically for every configuration of Table 2. Experiments for different configurations for the same system revealed that speed up of two orders of magnitude is possible with the compositional technique, while maintaining enough precision. The size of composition in the monolithic analysis is exponential in the depth of the hierarchy, due to a product construction (and it is also linear in the multiplication of sizes of all included standalone automata). In the compositional analysis, the depth of the hierarchy is constant (only two layers) and we only take a product of one root automaton with several constant size duration automata; this explains why the obtained speed-ups are so dramatic.

Configurations of Table 2	CFL	Comparison							
		concrete model		abstract model		monolithic model		compositional model	
		time	**size**	**time**	**size**	**time**	**size**	**time**	**size**
C1	2	No controller exists							
	1	94.20	290663	0.08	73	39.08	72608	0.09	76
C2	2	No controller exists							
	1	115.71	296524	0.11	107	71.41	83971	0.09	76
C3	2	No controller exists							
	1	Out of memory		0.14	242	Out of Memory		0.12	136
C4	2	Out of memory		0.25	712	Out of memory		0.19	439
	1			0.14	266			0.08	154
C5	2	Out of memory		0.25	712	Out of memory		0.19	439
	1			0.14	266			0.08	154
C6	2	No controller exists							
	1								
C7	2	Out of memory		0.25	712	Out of memory		0.20	439
	1			0.14	266			0.11	154
C8	2	Out of memory		0.15	420	Out of memory		0.14	278
	1			0.11	159	95.76	106960	0.09	101
C9	2	Out of memory		0.22	632	Out of memory		0.15	346
	1			0.14	234			0.10	124
C10	2	Out of memory		178.54	40668	Out of memory		64.60	18321
	1			73.32	14647			22.53	5868
C11	2	Out of memory		4.91	6274	Out of memory		5.05	5517
	1			1.65	2277			1.87	1783
C12	2	Out of memory		4.07	6272	Out of memory		3.18	4124
	1			1.65	2275			1.19	1338
C13	2	Out of memory		1.93	3639	Out of memory		3.18	4124
	1			0.81	1332			1.19	1338
C14	2	Out of memory		0.20	539	Out of memory		0.20	439
	1			0.14	204	78.12	118477	0.11	154
C15	2	Out of memory		0.15	431	Out of memory		0.21	530
	1			0.11	164	47.30	77548	0.13	183
C16	2	Out of memory		0.24	718	Out of memory		0.21	530
	1			0.14	270			0.13	183
C17	2	Out of memory		0.16	458	Out of memory		0.20	462
	1			0.12	173	59.26	109982	0.13	161
C18	2	Out of memory		0.16	485	Out of memory		0.20	453
	1			0.10	184	50.29	73914	0.12	158
C19	2	Out of memory		0.14	406	Out of memory		0.21	540
	1			0.10	154	45.14	84370	0.13	186
C20	2	Out of memory		0.14	358	94.07	179479	0.26	633
	1			0.09	135	34.14	63791	0.15	216

Table 3. Comparisons of the concrete, abstract, monolithic and compositional models with respect to synthesis time (in seconds) and the strategy size (in kilobytes)

The efficiency gains are primarily due to the coarse abstraction of safety and reachability properties of an arbitrarily large callee into a tiny duration automaton. Abstraction and compositional reasoning together might provide similar speed ups for TIOA in Section 2; and the restrictions that TPA impose on models allow one to automate the procedure. **OB2B)** For the monolithic models, the larger the difference between WCET and BECT the longer the analysis time, and the sparser the strategy, for example, configuration C1 versus configuration C2, C7 versus C14, and C15 versus C16. Unlike the other models, differences between the WCET and the corresponding BCET in the compositional model has no impact on the controller synthesis time or on the strategy size—for example, C1 versus C2, C7 versus C14, C12 versus C13, and C15 versus C16—because duration automata do not keep details regarding the best-case execution times. **OB3B)** The smaller the least common multiples of release periods the smaller state space, the shorter analysis time, and the more compact strategy, for instance, C2 versus C3, C8 versus C9, C9 versus C10, C10 versus C11, and so forth. **OB4B)** The least common multiples of the execution times have no clear impact on the analysis time or the size of the strategy, for example, C14 versus C15, C15 versus C17, C17 versus C18, C18 versus C19, C19 versus C20, and so forth. **OB5B)** Variations in the least common denominator of non-clock variables, such as different loads, do not have any significant impact on the analysis, for instance, C4 versus C5 and C5 versus C7. **OB5B)** Uppaal Tiga takes less time and generates a smaller strategy for a higher value for CFL, for instance, configurations C4, C5, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, and C20.

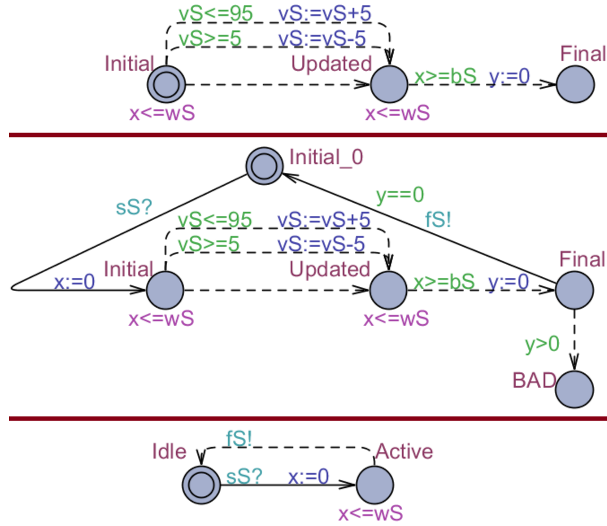


Fig. 15. Timed process automaton (in the top), standalone automaton (in the middle), and duration automaton (in the bottom) of task S of Section 2

Observations in the above match with the observations presented in Section 2.5. Depending on controller synthesis times or strategy sizes of Table 2 and Table 3, the following two relationships clearly hold: abstract ≤ monolithic ≤ concrete and compositional ≤ monolithic ≤ concrete. However, analyses results do not exhibit such explicit relationship between the abstract model and the compositional model. The compositional model has better outcomes than the abstract model for most

of the cases, for example, configurations C2 (for CFL 2), C3 (for CFL 2), C4, C5, C7, C8, C9, C10, C11, C12, C14, C16, C17 (for CFL 2), and C18.

3.5 Discussion

Timed process automata fall in the class of *TA with resources* [19] because of their ability to model dynamic behaviors, which is required when resource constraints do not permit one to activate all the components at the same time. More precisely, the model is a direct generalization of *task automata* [29], *dynamic networks of TA* [37], and *callable TA* [38]. These three variants model only closed systems, while TPA can model both closed and open systems. Task automata model only two layers (a scheduler and its tasks) of hierarchy, while TPA, dynamic networks of TA [37], and callable TA are able to model any numbers of hierarchies. Unlike TPA, none of them supports private communication, provides compositional modeling with reusable designs for different contexts, or supports automated state-space reduction technique.

Dynamic networks of continuous-time automata have also been studied in the context of hybrid automata [39,40]. These works model physical environments using differential equations, which restrict the kinds of environments that can be described. In practice, large differential equations make analyses unmanageable, or can only give statistical guarantees [40]. These works focus on system dynamics, and do not support private communication. Timed process automata can be considered as a member of the class of *TA with succinctness* [19] because they hide many design details from the designers to achieve succinctness (like *TA variants with urgency* [21,41,19]). Timed process automata are also timed game automata [7,15,13,9] because the new variant uses timed games for analysis.

4 Conclusions

In Section 2, we have presented the synthesis process using a mixed-criticality AMP system having a fault-intolerant criticality-unaware scheduler with fixed allocation. This includes two different design principles to model the problem using timed games, based on a selection of simplifications and abstractions. We compared the models for scalability, showing that solving the problem using strategy synthesis for timed games is feasible. We have observed that reducing action based synchronization, the state space, and especially shared states, improves efficiency of algorithms. Our reconfiguration services are distributed, and the synthesis process applies to mixed-criticality systems, both in symmetric and asymmetric scenarios. We demonstrated this on a case study from the automotive domain. This is the first case study applying timed games to the synthesis reconfiguration services for fault-tolerance.

In Section 3, we have presented TPA that captures dynamic activation and deactivation of continuous-time control processes and private communication among the active processes. We have provided a safety and reachability analysis technique for non-recursive well-formed TPA. We have also designed an

abstraction- and compositional reasoning-based state-space reduction technique for automated analysis of large industrial systems. Our analysis techniques can be applied in practice using any standard timed games solver such as Uppaal Tiga [17] and Synthia [18]. Timed process automata can model private communication and open systems. Moreover, TPA provide two important features for industrial dynamic open time-critical systems development: (i) compositional modeling with reusable designs for different contexts and (ii) automated state-space reduction technique.

Limitations and Future Works

Use of TA in the industry will be widespread if researchers can triumph over TA's state-space explosion problem and TA's realizability problem. Accurate TA implementability is getting more attention every day. Usually robustness analysis introduces larger state-spaces for example, Figure 3 of [42]. A study on the comparison and relation between these two problems—state-space explosion and robust analysis—of TA would be an interesting work for the research community. Our strong involvement with (automotive) industry and long experience in TA helped us to understand that state-space explosion is the biggest obstacle for TA. After our development of an automatable state-space reduction, the main challenge for timed (game) automata, therefore, is to improve computational efficiency of their symbolic semantics and data structures in a way that their computational complexity should be almost as expensive as their discrete-time counterpart.

Timed process automata allow compositional modeling with reuse by using channel-based dynamic renaming. One may explore this renaming process for other types of timed or hybrid or untimed auotmata to develop compositional modeling with reuse for the respective automata. One limitations for our compositional modeling with reuse is it handles only three representations. We, however, do not know other design aspects for which manual design alterations can be replaced by automated techniques. Investigating numerous large industrial models and surveying modeling experts might help one to find other design aspects that can be automated.Such type of investigation may also provide evidence that compositional modeling with reuse of TPA reduces modeling errors in practice. Findings of these investigations may encourage researchers to extend (timed process or other) automata's capability for compositional modeling with reuse.

Timed process automata facilitate automatable state-space reduction technique for timed games-based analysis of dynamic hierarchical systems. Theoretically manual state-space reduction may achieve similar or smaller state-spaces than automated state-space reduction. Even practically it is usually true for smaller systems for example, the comparisons in Table 3. However, efficiency of automated state-space reduction increases with depth of the control hierarchies in practice. Dynamic hierarchical systems with deep control hierarchies make up a small portion of all types of systems. Therefore, automated state-space reduction techniques for standard TIOA are much more important and desirable

than automated state-space reduction techniques for TPA. We strongly encourage researchers to develop an automated state-space reduction technique for standard TIOA. A similar but larger challenge is to develop a general automated state-space reduction technique for all types of TA.

This paper considers only location-based safety and location-based reachability properties of TPA. Investigation of other types of properties—including more general safety and reachability properties—may produce interesting outcomes. We use simple abstract model duration automata for our automatable state-space reduction technique. Others may prefer to use different kinds of abstract models for this purpose. Even for some scenarios or properties our duration automata might be too abstract to analyze. One may consider other state-space reduction techniques for TPA. For example, compositional model reduction of *discrete time systems (DES)* has been done by generalizing observers for deterministic DES to nondeterministic DES and characterizing using the join semilattice of compatible partitions of a transition system to achieve efficient algorithms [43,44].

It would be interesting to consider a model transformation from a subset of the *real-time π-calculus* [45,46] to TPA. This transformation might enable controllability analysis of π-calculus for open systems. The converse reduction from TPA to real-time π-calculus could also give several advantages: understanding TPA semantics in terms of the well-established π-calculus formalism, access to tools developed for real-time π-calculus [45], which might permit the analysis of recursive processes; it would also give a familiar automata-like syntax to π-calculus formalisms. It would also be relevant to minimize the number of subprocesses in controller synthesis. One may consider synthesis under this objective in the future, possibly by reduction to *priced/weighted TA* [47,48].

References

1. Kripke, S.: Semantical Considerations on Modal Logic. Acta Philosophica Fennica **16** (1963) 83–94
2. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming, New York, NY, USA, Springer-Verlag New York, Inc. (1990) 322–335
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (April 1994) 183–235
4. Ramchandani, C.: Analysis of asynchronous concurrent systems by timed Petri nets. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1974)
5. Ostroff, J.S.: Temporal Logic for Real Time Systems. John Wiley & Sons, Inc., New York, NY, USA (1989)
6. Waez, M.T.B., Wąsowski, A., Dingel, J., Rudie, K.: Synthesis of a reconfiguration service for mixed-criticality multi-core systems: An experience report. In Lanese, I., Madelaine, E., eds.: Formal Aspects of Component Software. Lecture Notes in Computer Science. Springer International Publishing (2015) 162–180
7. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems (an extended abstract). In: Symposium on Theoretical Aspects of Computer Science. (1995) 229–242

8. Henzinger, T.A., Kopke, P.W.: Discrete-time control for rectangular hybrid automata. Theoretical Computer Science **221** (June 1999) 369–392
9. David, A., Larsen, K.G., Legay, A., Nyman, U., Wąsowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control. HSCC '10, New York, NY, USA, ACM (2010) 91–100
10. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and Computation **111** (1994) 394–406
11. Henzinger, T.A., Manna, Z., Pnueli, A.: Timed transition systems. In de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G., eds.: Real-Time: Theory in Practice. Volume 600 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1992) 226–251
12. Kaynar, D.K., Lynch, N.A., Segala, R., Vaandrager, F.W.: The Theory of Timed I/O Automata. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers (2006)
13. Alfaro, L.d., Henzinger, T.A., Stoelinga, M.: Timed interfaces. In: Proceedings of the Second International Conference on Embedded Software. EMSOFT '02, London, UK, Springer-Verlag (2002) 108–122
14. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: Proceedings of the 5th IFAC Conference on System Structure and Control (SSC'98), Elsevier Science (July 1998) 469–474
15. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: The element of surprise in timed games. In Amadio, R., Lugiez, D., eds.: CONCUR 2003 - Concurrency Theory. Volume 2761 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2003) 144–158
16. David, A., Grunnet, J.D., Jessen, J.J., Larsen, K.G., Rasmussen, J.I.: Application of model-checking technology to controller synthesis. In Aichernig, B.K., de Boer, F.S., Bonsangue, M.M., eds.: Formal Methods for Components and Objects. Volume 6957 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 336–351
17. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Didier, L.: UPPAAL-Tiga: Time for playing games! In Damm, W., Hermanns, H., eds.: Computer Aided Verification. Volume 4590 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 121–125
18. Ehlers, R., Mattmüller, R., Peter, H.J.: Synthia: Verification and synthesis for timed automata. In Gopalakrishnan, G., Qadeer, S., eds.: Computer Aided Verification. Volume 6806 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 649–655
19. Waez, M.T.B., Dingel, J., Rudie, K.: A survey of timed automata for the development of real-time systems. Computer Science Review **9**(0) (2013) 1–26
20. Norström, C., Wall, A., Yi, W.: Timed automata as task models for event-driven systems. In: Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications. RTCSA '99, Washington, DC, USA, IEEE Computer Society (1999) 182–189
21. Bornot, S., Sifakis, J., Tripakis, S.: Modeling urgency in timed systems. In de Roever, W.P., Langmaack, H., Pnueli, A., eds.: Compositionality: The Significant Difference. Volume 1536 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1998) 103–129
22. Zhang, Y., Jiang, J.: Bibliographical review on reconfigurable fault-tolerant control systems. Annual Reviews in Control **32**(2) (2008) 229–252

23. Hwang, I., Kim, S., Kim, Y., Seah, C.E.: A survey of fault detection, isolation, and reconfiguration methods. IEEE Transactions on Control Systems Technology **18**(3) (May 2010) 636–653
24. Tripakis, S.: Fault diagnosis for timed automata. In Damm, W., Olderog, E.R., eds.: Formal Techniques in Real-Time and Fault-Tolerant Systems. Volume 2469 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2002) 205–221
25. Bouyer, P., Chevalier, F., D'Souza, D.: Fault diagnosis using timed automata. In Sassone, V., ed.: Foundations of Software Science and Computational Structures. Volume 3441 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 219–233
26. Waszniowski, L., Krákora, J., Hanzálek, Z.: Case study on distributed and fault tolerant system modeling based on timed automata. Journal of Systems and Software **82**(10) (October 2009) 1678–1694
27. Lv, M., Yi, W., Guan, N., Yu, G.: Combining abstract interpretation with model checking for timing analysis of multicore software. In: Proceedings of the 2010 31st IEEE Real-Time Systems Symposium. RTSS '10, Washington, DC, USA, IEEE Computer Society (2010) 339–349
28. Dalsgaard, A.E., Laarman, A., Larsen, K.G., Olesen, M.C., van de Pol, J.: Multi-core reachability for timed automata. In Jurdziński, M., Ničković, D., eds.: Formal Modeling and Analysis of Timed Systems. Volume 7595 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 91–106
29. Fersman, E., Krčál, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. International Journal of Information and Computation **205** (August 2007) 1149–1172
30. Socci, D., Poplavko, P., Bensalem, S., Bozga, M.: Modeling mixed-critical systems in real-time BIP. In: Proceedings of the Workshop on Real-Time Mixed Criticality Systems,. (2013)
31. Waez, M.T.B., Wąsowski, A., Dingel, J., Rudie, K.: A model for hierarchical open real-time systems. Technical report, Queen's University, ON (2016)
32. Waez, M.T.B., Wąsowski, A., Dingel, J., Rudie, K.: A model for industrial real-time systems. In D'Souza, D., Lal, A., Larsen, K.G., eds.: Verification, Model Checking, and Abstract Interpretation. Volume 8931 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2015) 153–171
33. Brihaye, T., Henzinger, T.A., Prabhu, V.S., Raskin, J.F.: Minimum-time reachability in timed games. In Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A., eds.: Automata, Languages and Programming. Volume 4596 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 825–837
34. Jurdziński, M., Laroussinie, F., Sproston, J.: Model checking probabilistic timed automata with one or two clocks. In: Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'07, Berlin, Heidelberg, Springer-Verlag (2007) 170–184
35. Cassez, F.: Timed games for computing WCET for pipelined processors with caches. In: Proceedings of the 2011 Eleventh International Conference on Application of Concurrency to System Design. ACSD '11, Washington, DC, USA, IEEE Computer Society (2011) 195–204
36. Gustavsson, A., Ermedahl, A., Lisper, B., Pettersson, P.: Towards wcet analysis of multicore architectures using UPPAAL. In Lisper, B., ed.: 10th International Workshop on Worst-Case Execution Time Analysis. Volume 15 of OASIcs., Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2010) 101–112

37. Campana, S., Spalazzi, L., Spegni, F.: Dynamic networks of timed automata for collaborative systems: A network monitoring case study. In: 2010 International Symposium on Collaborative Technologies and Systems. (May 2010) 113–122
38. Boudjadar, A., Vaandrager, F., Bodeveix, J.P., Filali, M.: Extending UPPAAL for the modeling and verification of dynamic real-time systems. In Arbab, F., Sirjani, M., eds.: Fundamentals of Software Engineering. Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 111–132
39. Göllü, A., Varaiya, P.: A dynamic network of hybrid automata. In: 5th annual conference on AI, simulation, and planning in high autonomy systems. (1994) 244–251
40. David, A., Larsen, K.G., Legay, A., Poulsen, D.B.: Statistical model checking of dynamic networks of stochastic hybrid automata. In Schneider, S., Treharne, H., eds.: Proceedings of the 13th International Workshop on Automated Verification of Critical Systems. Volume 10 of Electronic Communications of the EASST., Guildford, UK, EASST (2013)
41. Barbuti, R., Tesei, L.: Timed automata with urgent transitions. Acta Informatica **40** (March 2004) 317–347
42. Larsen, K.G., Legay, A., Traonouez, L.M., Wąsowski, A.: Robust specification of real time components. In: Proceedings of the 9th International Conference on Formal Modeling and Analysis of Timed Systems. FORMATS '11, Berlin, Heidelberg, Springer-Verlag (2011) 129–144
43. Lawford, M.: Model Reduction of Discrete Real-Time Systems. PhD thesis, Department of Electrical Computer Engineering, University of Toronto, Toronto, ON, Canada (1997)
44. Lawford, M., Wonham, W.M., Ostroff, J.S.: State-event observers for labeled transition systems. In: Proceedings of the 33rd IEEE Conference on Decision and Control. Volume 4. (December 1994) 3642–3648
45. Posse, E., Dingel, J.: Theory and implementation of a real-time extension to the π-calculus. In Hatcliff, J., Zucca, E., eds.: Formal Techniques for Distributed Systems. Volume 6117 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 125–139
46. Barakat, K., Kowalewski, S., Noll, T.: A native approach to modeling timed behavior in the Pi-calculus. In: 6th International Symposium on Theoretical Aspects of Software Engineering. (July 2012) 253–256
47. Alur, R., Torre, S.L., Pappas, G.J.: Optimal paths in weighted timed automata. In: Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control. HSCC '01, London, UK, Springer-Verlag (2001) 49–62
48. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.W.: Minimum-cost reachability for priced timed automata. In: Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control. HSCC '01, London, UK, Springer-Verlag (2001) 147–161

A Appendix for Section 2

Name	Type	Purpose
x	Clock	To record time passed since S was initialized
y	Clock	To record time passed since W was initialized
z	Clock	To record time passed since D was initialized
aS	Integer	To record the core which is currently assigned to execute S
aW	Integer	To record the core which is currently assigned to execute W
aD	Integer	To record the core which is currently assigned to execute D
iS	Boolean	To record whether S is initialized (1) or yet to initialize (0)
iW	Boolean	To record whether W is initialized (1) or yet to initialize (0)
iD	Boolean	To record whether D is initialized (1) or yet to initialize (0)
uS	Boolean	To record whether an update in S is performed (1) or not (0)
uW	Boolean	To record whether an update in W is performed (1) or not (0)
uD	Boolean	To record whether an update in D is performed (1) or not (0)
L1	Integer	To record the current worst possible loads on core_1
L2	Integer	To record the current worst possible loads on core_2
L3	Integer	To record the current worst possible loads on core_3
vS	Integer	To record the current value in the speedometer
sD	Integer	To record the current door state
F	Integer	To record the current total number of core failures

Table 4. Variables in the concrete model

Name	Type	Purpose
CFL	Constant	To store CFL
pS	Constant	To store release period of S
pW	Constant	To store release period of W
pD	Constant	To store release period of D
wS	Constant	To store the WCET of S
wW	Constant	To store the WCET of W
wD	Constant	To store the WCET of D
bS	Constant	To store the BCET of S
bW	Constant	To store the BCET of W
bD	Constant	To store the BCET of D
lS1	Constant	To store the worst-case load of S on core_1
lS2	Constant	To store the worst-case load of S on core_2
lS3	Constant	To store the worst-case load of S on core_3
lW1	Constant	To store the worst-case load of W on core_1
lW2	Constant	To store the worst-case load of W on core_2
lW3	Constant	To store the worst-case load of W on core_3
lD1	Constant	To store the worst-case load of D on core_1
lD2	Constant	To store the worst-case load of D on core_2
lD3	Constant	To store the worst-case load of D on core_3
Limit	Constant	To store the load limit of every core

Table 5. Constants in the concrete model

Name	From	To	Purpose
iS1	$core_1$	$core_1$.S	To initialize S on $core_1$
iS2	$core_2$	$core_2$.S	To initialize S on $core_2$
iS3	$core_3$	$core_3$.S	To initialize S on $core_3$
iW1	$core_1$	$core_1$.W	To initialize W on $core_1$
iW2	$core_2$	$core_2$.W	To initialize W on $core_2$
iW3	$core_3$	$core_3$.W	To initialize W on $core_3$
iD1	$core_1$	$core_1$.D	To initialize D on $core_1$
iD2	$core_2$	$core_2$.D	To initialize D on $core_2$
iD3	$core_3$	$core_3$.D	To initialize D on $core_3$
rS1	service	$core_1$.S	To resume S on $core_1$
rS2	service	$core_2$.S	To resume S on $core_2$
rS3	service	$core_3$.S	To resume S on $core_3$
rW1	service	$core_1$.W	To resume W on $core_1$
rW2	service	$core_2$.W	To resume W on $core_2$
rW3	service	$core_3$.W	To resume W on $core_3$
rD1	service	$core_1$.D	To resume D on $core_1$
rD2	service	$core_2$.D	To resume D on $core_2$
rD3	service	$core_3$.D	To resume D on $core_3$
kS1	$core_1$, service	$core_1$.S	To kill S on $core_1$
kS2	$core_2$, service	$core_2$.S	To kill S on $core_2$
kS3	$core_3$, service	$core_3$.S	To kill S on $core_3$
kW1	$core_1$, service	$core_1$.W	To kill W on $core_1$
kW2	$core_2$, service	$core_2$.W	To kill W on $core_2$
kW3	$core_3$, service	$core_3$.W	To kill W on $core_3$
kD1	$core_1$, service	$core_1$.D	To kill D on $core_1$
kD2	$core_2$, service	$core_2$.D	To kill D on $core_2$
kD3	$core_3$, service	$core_3$.D	To kill D on $core_3$

Table 6. Actions in the concrete model (part 1)

Name	From	To	Purpose
tS1	$core_1$.S	$core_1$	To terminate S on $core_1$
tS2	$core_2$.S	$core_2$	To terminate S on $core_2$
tS3	$core_3$.S	$core_3$	To terminate S on $core_3$
tW1	$core_1$.W	$core_1$	To terminate W on $core_1$
tW2	$core_2$.W	$core_2$	To terminate W on $core_2$
tW3	$core_3$.W	$core_3$	To terminate W on $core_3$
tD1	$core_1$.D	$core_1$	To terminate D on $core_1$
tD2	$core_2$.D	$core_2$	To terminate D on $core_2$
tD3	$core_3$.D	$core_3$	To terminate D on $core_3$
mSW	$core_1$, $core_2$	service	To inform that it is assigned to execute S and W
mSD	$core_1$, $core_3$	service	To inform that it is assigned to execute S and D
mWD	$core_2$, $core_3$	service	To inform that it is assigned to execute W and D
mS	$core_1$	service	To inform that it is assigned to execute S
mW	$core_2$	service	To inform that it is assigned to execute W
mD	$core_3$	service	To inform that it is assigned to execute D

Table 7. Actions in the concrete model (part 2)

Name	Type	Purpose
x	Clock	To record time passed since S was initialized
y	Clock	To record time passed since W was initialized
z	Clock	To record time passed since D was initialized
aS1	Boolean	To record whether core_1 is currently assigned (1) to execute S or not (0)
aW1	Boolean	To record whether core_1 is currently assigned (1) to execute W or not (0)
aD1	Boolean	To record whether core_1 is currently assigned (1) to execute D or not (0)
aS2	Boolean	To record whether core_2 is currently assigned (1) to execute S or not (0)
aW2	Boolean	To record whether core_2 is currently assigned (1) to execute W or not (0)
aD2	Boolean	To record whether core_2 is currently assigned (1) to execute D or not (0)
aS3	Boolean	To record whether core_3 is currently assigned (1) to execute S or not (0)
aW3	Boolean	To record whether core_3 is currently assigned (1) to execute W or not (0)
aD3	Boolean	To record whether core_3 is currently assigned (1) to execute D or not (0)
iS	Boolean	To record whether S is initialized (1) or yet to initialize (0)
iW	Boolean	To record whether W is initialized (1) or yet to initialize (0)
iD	Boolean	To record whether D is initialized (1) or yet to initialize (0)
L1	Integer	To record the current worst possible loads on core_1
L2	Integer	To record the current worst possible loads on core_2
L3	Integer	To record the current worst possible loads on core_3
F	Integer	To record the current total number of core failures

Table 8. Variables in the abstract model

Name	Type	Purpose
CFL	Constant	To store CFL
pS	Constant	To store release period of S
pW	Constant	To store release period of W
pD	Constant	To store release period of D
wS	Constant	To store the WCET of S
wW	Constant	To store the WCET of W
wD	Constant	To store the WCET of D
bS	Constant	To store the BCET of S
bW	Constant	To store the BCET of W
bD	Constant	To store the BCET of D
lS1	Constant	To store the worst-case load of S on core_1
lS2	Constant	To store the worst-case load of S on core_2
lS3	Constant	To store the worst-case load of S on core_3
lW1	Constant	To store the worst-case load of W on core_1
lW2	Constant	To store the worst-case load of W on core_2
lW3	Constant	To store the worst-case load of W on core_3
lD1	Constant	To store the worst-case load of D on core_1
lD2	Constant	To store the worst-case load of D on core_2
lD3	Constant	To store the worst-case load of D on core_3
Limit	Constant	To store the load limit of every core

Table 9. Constants in the abstract model

```
void initialize(int[1,3] task, int[1,3] core)
{
        if (task==S && core==1)          {L1=L1+lS1;      iS=1;}
        else if (task==S && core==2)     {L2=L2+lS2;      iS=1;}
        else if (task==S && core==3)     {L3=L3+lS3;      iS=1;}
        else if (task==W && core==1)     {L1=L1+lW1;      iW=1;}
        else if (task==W && core==2)     {L2=L2+lW2;      iW=1;}
        else if (task==W && core==3)     {L3=L3+lW3;      iW=1;}
        else if (task==D && core==1)     {L1=L1+lD1;      iD=1;}
        else if (task==D && core==2)     {L2=L2+lD2;      iD=1;}
        else if (task==D && core==3)     {L3=L3+lD3;      iD=1;}
}
void terminate(int[1,3] task, int[1,3] core)
{
        if (task==S && core==1)          {L1=L1-lS1;      iS=0;}
        else if (task==S && core==2)     {L2=L2-lS2;      iS=0;}
        else if (task==S && core==3)     {L3=L3-lS3;      iS=0;}
        else if (task==W && core==1)     {L1=L1-lW1;      iW=0;}
        else if (task==W && core==2)     {L2=L2-lW2;      iW=0;}
        else if (task==W && core==3)     {L3=L3-lW3;      iW=0;}
        else if (task==D && core==1)     {L1=L1-lD1;      iD=0;}
        else if (task==D && core==2)     {L2=L2-lD2;      iD=0;}
        else if (task==D && core==3)     {L3=L3-lD3;      iD=0;}
}
```

Fig. 16. Functions `initialize` and `terminate` in the concrete model

Fig. 17. Automaton core_1.`W` in the concrete model

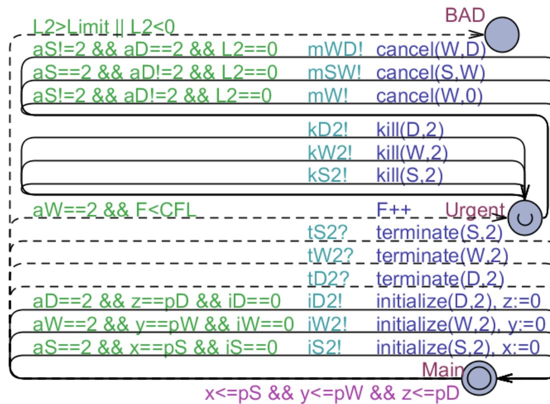


Fig. 18. Automaton core_2 in the concrete model

```
void kill(int[1,3] task, int[1,3] core)
{
        if (task==S && core==1)         L1=L1-lS1;
        else if (task==S && core==2)    L2=L2-lS2;
        else if (task==S && core==3)    L3=L3-lS3;
        else if (task==W && core==1)    L1=L1-lW1;
        else if (task==W && core==2)    L2=L2-lW2;
        else if (task==W && core==3)    L3=L3-lW3;
        else if (task==D && core==1)    L1=L1-lD1;
        else if (task==D && core==2)    L2=L2-lD2;
        else if (task==D && core==3)    L3=L3-lD3;
}
void cancel(int[0,3] task1, int[0,3]  task2)
{
        if (task1==S && task2==0)       aS=0;
        else if (task1==W && task2==0)  aW=0;
        else if (task1==D && task2==0)  aD=0;
        else if (task1==0 && task2==S)  aS=0;
        else if (task1==0 && task2==W)  aW=0;
        else if (task1==0 && task2==D)  aD=0;
        else if (task1==S && task2==W)  {aS=0;  aW=0;}
        else if (task1==W && task2==S)  {aS=0;  aW=0;}
        else if (task1==S && task2==D)  {aS=0;  aD=0;}
        else if (task1==D && task2==S)  {aS=0;  aD=0;}
        else if (task1==W && task2==D)  {aW=0;  aD=0;}
        else if (task1==D && task2==W)  {aW=0;  aD=0;}
}
```

Fig. 19. Functions `kill`, and `cancel` in the concrete model

Fig. 20. Automaton core_2.W in the concrete model

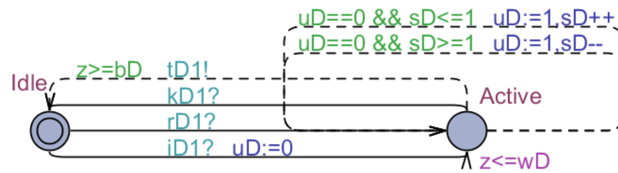


Fig. 21. Automaton core_1.D in the concrete model

```
void resume(int[1,3] task, int[1,3] core)
{
        if (task==S && core==1)          {aS=1;  L1=L1+lS1;}
        else if (task==S && core==2)     {aS=2;  L2=L2+lS2;}
        else if (task==S && core==3)     {aS=3;  L3=L3+lS3;}
        else if (task==W && core==1)     {aW=1;  L1=L1+lW1;}
        else if (task==W && core==2)     {aW=2;  L2=L2+lW2;}
        else if (task==W && core==3)     {aW=3;  L3=L3+lW3;}
        else if (task==D && core==1)     {aD=1;  L1=L1+lD1;}
        else if (task==D && core==2)     {aD=2;  L2=L2+lD2;}
        else if (task==D && core==3)     {aD=3;  L3=L3+lD3;}
}
void reassign(int[1,3] task, int[1,3] core)
{
        if (task==S && core==1)          aS=1;
        else if (task==S && core==2)     aS=2;
        else if (task==S && core==3)     aS=3;
        else if (task==W && core==1)     aW=1;
        else if (task==W && core==2)     aW=2;
        else if (task==W && core==3)     aW=3;
        else if (task==D && core==1)     aD=1;
        else if (task==D && core==2)     aD=2;
        else if (task==D && core==3)     aD=3;
}
```

Fig. 22. Functions `resume` and `reassign` in the concrete model

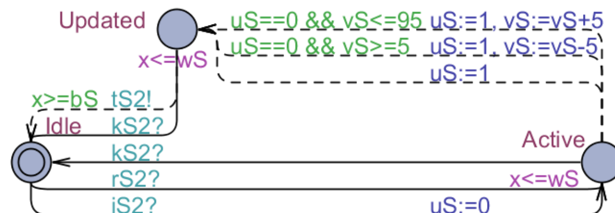


Fig. 23. Automaton $core_2$.S in the concrete model

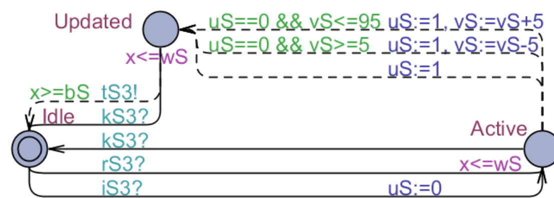


Fig. 24. Automaton $core_3$.S in the concrete model

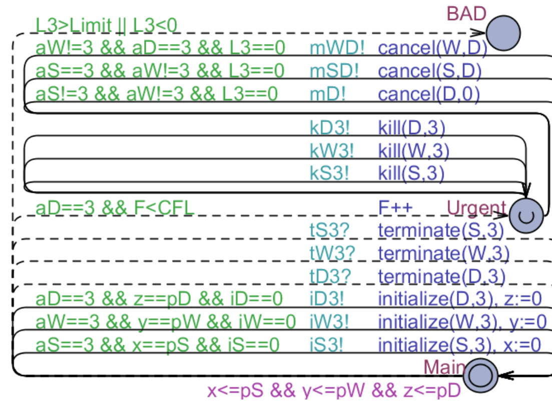


Fig. 25. Automaton core_3 in the $\mathsf{concrete}$ model

Fig. 26. Automaton $\mathsf{core}_3.\mathtt{W}$ in the $\mathsf{concrete}$ model

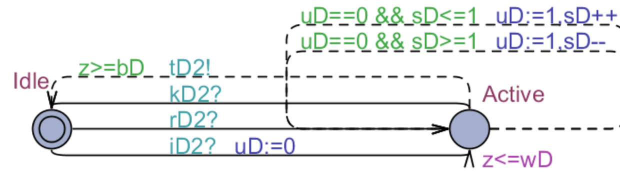


Fig. 27. Automaton $\mathsf{core}_2.\mathtt{D}$ in the $\mathsf{concrete}$ model

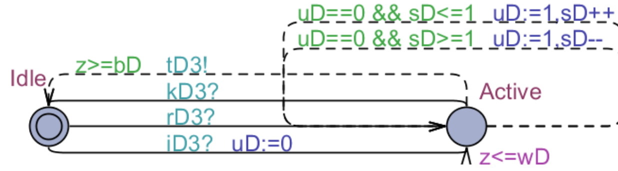


Fig. 28. Automaton core_3.D in the concrete model

```
void reallocate(int[1,3] task, int[1,3] core){
        if (task==S && core==1)         {aS1=1;         L1=L1+iS*lS1;
                if (aS2==1)             {aS2=0;         L2=L2-iS*lS2;}
                else if (aS3==1)        {aS3=0;         L3=L3-iS*lS3;}}
        else if (task==S && core==2)    {aS2=1;         L2=L2+iS*lS2;
                 if (aS1==1)            {aS1=0;         L1=L1-iS*lS1;}
                else if (aS3==1)        {aS3=0;         L3=L3-iS*lS3;}}
        else if (task==S && core==3)    {aS3=1;         L3=L3+iS*lS3;
                if (aS1==1)             {aS1=0;         L1=L1-iS*lS1;}
                else if (aS2==1)        {aS2=0;         L2=L2-iS*lS2;}}
        else if (task==W && core==1)    {aW1=1;         L1=L1+iW*lW1;
                 if (aW2==1)            {aW2=0;         L2=L2-iW*lW2;}
                else if (aW3==1)        {aW3=0;         L3=L3-iW*lW3;}}
        else if (task==W && core==2)    {aW2=1;         L2=L2+iW*lW2;
                 if (aW1==1)            {aW1=0;         L1=L1-iW*lW1;}
                else if (aW3==1)        {aW3=0;         L3=L3-iW*lW3;}}
        else if (task==W && core==3)    {aW3=1;         L3=L3+iW*lW3;
                 if (aW1==1)            {aW1=0;         L1=L1-iW*lW1;}
                else if (aW2==1)        {aW2=0;         L2=L2-iW*lW2;}}
        else if (task==D && core==1)    {aD1=1;         L1=L1+iD*lD1;
                 if (aD2==1)            {aD2=0;         L2=L2-iD*lD2;}
                else if (aD3==1)        {aD3=0;         L3=L3-iD*lD3;}}
        else if (task==D && core==2)    {aD2=1;         L2=L2+iD*lD2;
                 if (aD1==1)            {aD1=0;         L1=L1-iD*lD1;}
                else if (aD3==1)        {aD3=0;         L3=L3-iD*lD3;}}
        else if (task==D && core==3)    {aD3=1;         L3=L3+iD*lD3;
                if (aD1==1)             {aD1=0;         L1=L1-iD*lD1;}
                else if (aD2==1)        {aD2=0;         L2=L2-iD*lD2;}}}
```

Fig. 29. Function `reallocate` in the abstract model

```
void initializeA(int[1,3] task)
{
        if (task==S)            {L1=L1+lS1*aS1; L2=L2+lS2*aS2; L3=L3+lS3*aS3;  iS=1;}
        else if (task==W)       {L1=L1+lW1*aW1; L2=L2+lW2*aW2; L3=L3+lW3*aW3; iW=1;}
        else if (task==D)       {L1=L1+lD1*aD1; L2=L2+lD2*aD2; L3=L3+lD3*aD3; iD=1;}
}

void terminateA(int[1,3] task)
{
        if (task==S)            {L1=L1-lS1*aS1; L2=L2-lS2*aS2; L3=L3-lS3*aS3; iS=0;}
        else if (task==W)       {L1=L1-lW1*aW1; L2=L2-lW2*aW2; L3=L3-lW3*aW3; iW=0;}
        else if (task==D)       {L1=L1-lD1*aD1; L2=L2-lD2*aD2; L3=L3-lD3*aD3; iD=0;}
}
```

Fig. 30. Functions `initializeA` and `terminateA` in the `abstract` model

B Appendix for Section 3

Sample I/O

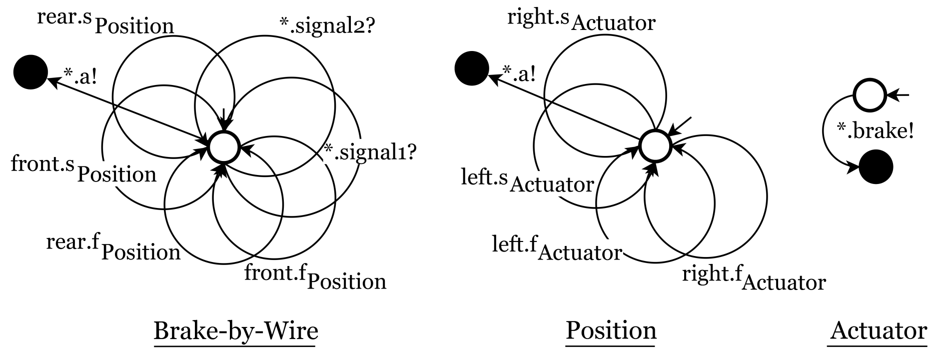


Fig. 31. The Brake-by-Wire system

Input: timed process automaton **Actuator** creation

1. Name: Actuator
2. Input Actions: ∅
3. Output Actions: {Brake}
4. Callees: ∅
5. Clocks: ∅
6. Channels: ∅
7. Locations: $l0$, $l1$
8. Initial Location: $l0$
9. Final Location: $l1$
10. Invariant: $l0$:*true*, $l1$:*true*
11. Edges:
    - New edge:
        - Source Location: $l0$
        - Action: brake
        - Channel: $*$
        - Clock Constraint: ∅
        - Clock Resets: ∅
        - Destination Location: $l1$

Input: timed process automaton **Position** creation

1. Name: Position
2. Input Actions: ∅
3. Output Actions: {a}
4. Callees: Actuator
5. Clocks: ∅
6. Channels: {right, left}
7. Locations: $l0$, $l1$
8. Initial Location: $l0$
9. Final Location: $l1$
10. Invariant: $l0$:*true*, $l1$:*true*

11. Edges:
    - New edge:
        - Source Location: $l0$
        - Action: a
        - Channel: $*$
        - Clock Constraint: $\emptyset$
        - Clock Resets: $\emptyset$
        - Destination Location: $l1$
    - New edge:
        - Source Location: $l0$
        - Action: $s_{Actuator}$
        - Channel: right
        - Clock Constraint: $\emptyset$
        - Clock Resets: $\emptyset$
        - Destination Location: $l0$
    - New edge:
        - Source Location: $l0$
        - Action: $s_{Actuator}$
        - Channel: left
        - Clock Constraint: $\emptyset$
        - Clock Resets: $\emptyset$
        - Destination Location: $l0$
    - New edge:
        - Source Location: $l0$
        - Action: $f_{Actuator}$
        - Channel: right
        - Clock Constraint: $\emptyset$
        - Clock Resets: $\emptyset$
        - Destination Location: $l0$
    - New edge:
        - Source Location: $l0$
        - Action: $f_{Actuator}$
        - Channel: left
        - Clock Constraint: $\emptyset$
        - Clock Resets: $\emptyset$
        - Destination Location: $l0$

Input: timed process automaton Brake-by-Wire creation

1. Name: Brake-by-Wire
2. Input Actions: $\{signal1, signal2\}$
3. Output Actions: {a}
4. Callees: Position
5. Clocks: $\emptyset$
6. Channels: {front, rear}
7. Locations: $l0$, $l1$
8. Initial Location: $l0$
9. Final Location: $l1$
10. Invariant: $l0$:$true$, $l1$:$true$
11. Edges:
    - New edge:
        - Source Location: $l0$
        - Action: a
        - Channel: $*$
        - Clock Constraint: $\emptyset$
        - Clock Resets: $\emptyset$
        - Destination Location: $l1$
    - New edge:
        - Source Location: $l0$
        - Action: $s_{Position}$
        - Channel: front
        - Clock Constraint: $\emptyset$
        - Clock Resets: $\emptyset$
        - Destination Location: $l0$
    - New edge:
        - Source Location: $l0$
        - Action: $s_{Position}$

  - Channel: rear
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: $\mathsf{f}_{\mathsf{Position}}$
  - Channel: front
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: $\mathsf{f}_{\mathsf{Position}}$
  - Channel: rear
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: signal1
  - Channel: $*$
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: signal2
  - Channel: $*$
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$

Output: timed process automaton **Actuator** display

- $Name = \mathsf{Actuator}$
- $Locations = \{l0 : true, l1 : true\}$
- $initialLocation = l0$
- $finalLocation = l1$
- $Edges = \{(l0, brake!, *, \emptyset, \emptyset, l1)\}$

Output: timed process automaton **Position** display

- $Name = \mathsf{Position}$
- $Locations = \{l0 : true, l1 : true\}$
- $initialLocation = l0$
- $finalLocation = l1$
- $Edges = \{(l0, a!, *, \emptyset, \emptyset, l1), (l0, \mathsf{s}_{\mathsf{Actuator}}, \mathsf{right}, \emptyset, \emptyset, l0), (l0, \mathsf{s}_{\mathsf{Actuator}}, \mathsf{left}, \emptyset, \emptyset, l0),$ $(l0, \mathsf{f}_{\mathsf{Actuator}}, \mathsf{right}, \emptyset, \emptyset, l0), (l0, \mathsf{f}_{\mathsf{Actuator}}, \mathsf{left}, \emptyset, \emptyset, l0)\}$

Output: timed process automaton **Brake-by-Wire** display

- $Name = \mathsf{Brake\text{-}by\text{-}Wire}$
- $Locations = \{l0 : true, l1 : true\}$
- $initialLocation = l0$
- $finalLocation = l1$
- $Edges = \{(l0, a!, *, \emptyset, \emptyset, l1), (l0, \mathsf{s}_{\mathsf{Position}}, \mathsf{right}, \emptyset, \emptyset, l0), (l0, \mathsf{s}_{\mathsf{Position}}, \mathsf{left}, \emptyset, \emptyset, l0),$ $(l0, \mathsf{f}_{\mathsf{Position}}, \mathsf{right}, \emptyset, \emptyset, l0), (l0, \mathsf{f}_{\mathsf{Position}}, \mathsf{left}, \emptyset, \emptyset, l0), (l0, \mathsf{signal1}?, *, \emptyset, \emptyset, l0),$ $(l0, \mathsf{signal2}?, *, \emptyset, \emptyset, l0)\}$

Output: a monolithic analysis model of timed process automaton Actuator

- Timed game analysis of $\mathsf{root}(P_0)$
  - $\mathsf{root}(P_0)$ is
    - $Identifier = P_0$
    - $\mathsf{tpa}(P_0)$=Actuator
    - $\mathsf{channel}(P_0)=\perp$
    - $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
    - $initialLocation = l0$
    - $Edges = \{(l0, P_0.*.brake!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l1, \perp.\mathsf{f}_{\mathsf{Actuator}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0.*.u?, n = 0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD)\}$

Output: a monolithic analysis model of timed process automaton Position

- Timed game analysis of $\mathsf{root}(P_0)||\mathsf{standalone}(P_1)||\mathsf{standalone}(P_2)$
  - $\mathsf{root}(P_0)$ is
    - $Identifier = P_0$
    - $\mathsf{tpa}(P_0)$=Position
    - $\mathsf{channel}(P_0)=\perp$
    - $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
    - $initialLocation = l0$
    - $Edges = \{(l0, P_0.*.a!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l0, P_0.\mathsf{right}.\mathsf{s}_{\mathsf{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0),$
      $(l0, P_0.\mathsf{left}.\mathsf{s}_{\mathsf{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_0\mathsf{right}.\mathsf{f}_{\mathsf{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0),$
      $(l0, P_0.\mathsf{left}.\mathsf{f}_{\mathsf{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, \perp.\mathsf{f}_{\mathsf{Position}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0.*.u?, n = 0 \wedge x^{P_0} >$
      $0, \emptyset, \emptyset, BAD)\}$
  - $\mathsf{standalone}(P_1)$ is
    - $Identifier = P_1$
    - $\mathsf{tpa}(P_1)$=Actuator
    - $\mathsf{channel}(P_1)=P_0.\mathsf{right}$
    - $Locations = \{l0 : true, l1 : true, l_0^{P_1} : true, BAD : true\}$
    - $initialLocation = l_0^{P_1}$
    - $Edges = \{(l0, P_1*.brake!, \emptyset, \emptyset, \{x^{P_1}\}, l1), (l1, P_0.\mathsf{right}.\mathsf{f}_{\mathsf{Actuator}}!, n = 0 \wedge x^{P_1} = 0, \emptyset, \emptyset, l_0^{P_1}), (l_0^{P_1}, P_0.\mathsf{right}.\mathsf{s}_{\mathsf{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_1*$
      $.u?, n = 0 \wedge x^{P_1} > 0, \emptyset, \emptyset, BAD)\}$
  - $\mathsf{standalone}(P_2)$ is
    - $Identifier = P_2$
    - $\mathsf{tpa}(P_2)$=Actuator
    - $\mathsf{channel}(P_2)=P_0.\mathsf{left}$
    - $Locations = \{l0 : true, l1 : true, l_0^{P_2} : true, BAD : true\}$
    - $initialLocation = l_0^{P_2}$
    - $Edges = \{(l0, P_2*.brake!, \emptyset, \emptyset, \{x^{P_2}\}, l1), (l1, P_0.\mathsf{left}.\mathsf{f}_{\mathsf{Actuator}}!, n = 0 \wedge x^{P_2} = 0, \emptyset, \emptyset, l_0^{P_2}), (l_0^{P_2}, P_0.\mathsf{left}.\mathsf{s}_{\mathsf{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_2*$
      $.u?, n = 0 \wedge x^{P_2} > 0, \emptyset, \emptyset, BAD)\}$

Output: a monolithic analysis model of timed process automaton Brake-by-Wire

- Timed game analysis of $\mathsf{root}(P_0)||\mathsf{standalone}(P_1)||\mathsf{standalone}(P_2)||\mathsf{standalone}(P_3)||$
  $\mathsf{standalone}(P_4)||\mathsf{standalone}(P_5)||\mathsf{standalone}(P_6)$
  - $\mathsf{root}(P_0)$ is
    - $Identifier = P_0$
    - $\mathsf{tpa}(P_0)$=Brake-by-Wire
    - $\mathsf{channel}(P_0)=\perp$
    - $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
    - $initialLocation = l0$
    - $Edges = \{(l0, P_0.*.a!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l0, P_0.\mathsf{front}.\mathsf{s}_{\mathsf{Position}}!, \emptyset, \{n++\}, \emptyset, l0),$
      $(l0, P_0.\mathsf{rear}.\mathsf{s}_{\mathsf{Position}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_0\mathsf{front}.\mathsf{f}_{\mathsf{Position}}?, \emptyset, \{n--\}, \emptyset, l0),$
      $(l0, P_0.\mathsf{rear}.\mathsf{f}_{\mathsf{Position}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, \perp.\mathsf{f}_{\mathsf{Brake\text{-}by\text{-}Wire}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0.*.u?, n =$
      $0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD), (l0, P_0.*.\mathsf{signal1}?, \emptyset, \emptyset, \emptyset, l0),$
      $(l0, P_0.*.\mathsf{signal2}?, \emptyset, \emptyset, \emptyset, l0)\}$

- standalone(P_1) is
  - $Identifier = P_1$
  - tpa(P_1)=Position
  - channel(P_1)=P_0.front
  - $Locations = \{l0 : true, l1 : true, l_0^{P_1} : true, BAD : true\}$
  - $initialLocation = l_0^{P_1}$
  - $Edges = \{(l0, P_1.*.a!, \emptyset, \emptyset, \{x^{P_1}\}, l1), (l0, P_1.\text{right.s}_{\text{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0),$ $(l0, P_1.\text{left.s}_{\text{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_1\text{right.f}_{\text{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0),$ $(l0, P_1.\text{left.f}_{\text{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, P_0.\text{front.f}_{\text{Position}}!, n = 0 \wedge x^{P_1} = 0, \emptyset, \emptyset, l_0^{P_1}),$ $(l_0^{P_1}, P_0.\text{front.s}_{\text{Position}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_1.*.u?, n = 0 \wedge x^{P_1} > 0, \emptyset, \emptyset, BAD)\}$
- standalone(P_2) is
  - $Identifier = P_2$
  - tpa(P_2)=Position item channel(P_2)=P_0.rear
  - $Locations = \{l0 : true, l1 : true, l_0^{P_2} : true, BAD : true\}$
  - $initialLocation = l_0^{P_2}$
  - $Edges = \{(l0, P_2.*.a!, \emptyset, \emptyset, \{x^{P_2}\}, l1), (l0, P_2.\text{right.s}_{\text{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0),$ $(l0, P_2.\text{left.s}_{\text{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_2\text{right.f}_{\text{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0),$ $(l0, P_2.\text{left.f}_{\text{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, P_0.\text{rear.f}_{\text{Position}}!, n = 0 \wedge x^{P_2} = 0, \emptyset, \emptyset, l_0^{P_2}), (l_0^{P_2}, P_0.\text{rear.s}_{\text{Position}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_2*$ $.u?, n = 0 \wedge x^{P_2} > 0, \emptyset, \emptyset, BAD)\}$
- standalone(P_3) is
  - $Identifier = P_3$
  - tpa(P_3)=Actuator
  - channel(P_3)=P_1.right
  - $Locations = \{l0 : true, l1 : true, l_0^{P_3} : true, BAD : true\}$
  - $initialLocation = l_0^{P_3}$
  - $Edges = \{(l0, P_3*brake!, \emptyset, \emptyset, \{x^{P_3}\}, l1), (l1, P_1.\text{right.f}_{\text{Actuator}}!, n = 0 \wedge x^{P_3} = 0, \emptyset, \emptyset, l_0^{P_3}), (l_0^{P_3}, P_1.\text{right.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_3*$ $.u?, n = 0 \wedge x^{P_3} > 0, \emptyset, \emptyset, BAD)\}$
- standalone(P_4) is
  - $Identifier = P_4$
  - tpa(P_4)=Actuator
  - channel(P_4)=P_1.left
  - $Locations = \{l0 : true, l1 : true, l_0^{P_4} : true, BAD : true\}$
  - $initialLocation = l_0^{P_4}$
  - $Edges = \{(l0, P_4*brake!, \emptyset, \emptyset, \{x^{P_4}\}, l1), (l1, P_1.\text{left.f}_{\text{Actuator}}!, n = 0 \wedge x^{P_4} = 0, \emptyset, \emptyset, l_0^{P_4}), (l_0^{P_4}, P_1.\text{left.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_4*$ $.u?, n = 0 \wedge x^{P_4} > 0, \emptyset, \emptyset, BAD)\}$
- standalone(P_5) is
  - $Identifier = P_5$
  - tpa(P_5)=Actuator
  - channel(P_5)=P_2.right
  - $Locations = \{l0 : true, l1 : true, l_0^{P_5} : true, BAD : true\}$
  - $initialLocation = l_0^{P_5}$
  - $Edges = \{(l0, P_5*brake!, \emptyset, \emptyset, \{x^{P_5}\}, l1), (l1, P_2.\text{right.f}_{\text{Actuator}}!, n = 0 \wedge x^{P_5} = 0, \emptyset, \emptyset, l_0^{P_5}), (l_0^{P_5}, P_2.\text{right.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_5*$ $.u?, n = 0 \wedge x^{P_5} > 0, \emptyset, \emptyset, BAD)\}$
- standalone(P_6) is
  - $Identifier = P_6$
  - tpa(P_6)=Actuator
  - channel(P_6)=P_2.left
  - $Locations = \{l0 : true, l1 : true, l_0^{P_6} : true, BAD : true\}$
  - $initialLocation = l_0^{P_6}$
  - $Edges = \{(l0, P_6*brake!, \emptyset, \emptyset, \{x^{P_6}\}, l1), (l1, P_2.\text{left.f}_{\text{Actuator}}!, n = 0 \wedge x^{P_6} = 0, \emptyset, \emptyset, l_0^{P_6}), (l_0^{P_6}, P_2.\text{left.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_6*$ $.u?, n = 0 \wedge x^{P_6} > 0, \emptyset, \emptyset, BAD)\}$

Output: a compositional analysis model of timed process automaton Actuator

– Timed game analysis of root(P_0)
- root(P_0) is
  - $Identifier = P_0$
  - tpa(P_0)=Actuator

* channel(P_0)=$\perp$
* $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
* $initialLocation = l0$
* $Edges = \{(l0, P_0.*.brake!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l1, \perp.\mathsf{f}_{\mathsf{Actuator}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0.*.u?, n = 0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD)\}$

Output: a compositional analysis model of timed process automaton Position

- Timed game analysis of root(P_0)||duration(P_1)||duration(P_2)
  - root(P_0) is
    * $Identifier = P_0$
    * tpa(P_0)=Position
    * channel(P_0)=$\perp$
    * $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
    * $initialLocation = l0$
    * $Edges = \{(l0, P_0.*.a!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l0, P_0.\mathsf{right.s}_{\mathsf{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0),$ $(l0, P_0.\mathsf{left.s}_{\mathsf{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_0\mathsf{right.f}_{\mathsf{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0),$ $(l0, P_0.\mathsf{left.f}_{\mathsf{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, \perp.\mathsf{f}_{\mathsf{Position}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0.*.u?, n = 0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD)\}$
  - duration(P_1) is
    * $Identifier = P_1$
    * tpa(P_1)=Actuator
    * channel(P_1)=P_0.right
    * *The WCET (input from the game analyzer)=Unknown*
    * $Locations = \{l_0^{P_1} : true, l_1^{P_1} : true\}$
    * $initialLocation = l_0^{P_1}$
    * $Edges = \{(l_1^{P_1}, P_0.\mathsf{right.f}_{\mathsf{Actuator}}!, \emptyset, \emptyset, \emptyset, l_0^{P_1}), (l_0^{P_1}, P_0.\mathsf{right.s}_{\mathsf{Actuator}}?, \emptyset, \emptyset, \emptyset, l_1^{P_1})\}$
  - duration(P_2) is
    * $Identifier = P_2$
    * tpa(P_2)=Actuator
    * channel(P_2)=P_0.left
    * *The WCET (input from the game analyzer)=Unknown*
    * $Locations = \{l_0^{P_2} : true, l_1^{P_2} : true\}$
    * $initialLocation = l_0^{P_2}$
    * $Edges = \{(l_1^{P_2}, P_0.\mathsf{left.f}_{\mathsf{Actuator}}!, \emptyset, \emptyset, \emptyset, l_0^{P_2}), (l_0^{P_2}, P_0.\mathsf{left.s}_{\mathsf{Actuator}}?, \emptyset, \emptyset, \emptyset, l_1^{P_2})\}$

Output: a compositional analysis model of timed process automaton Brake-by-Wire

- Timed game analysis of root(P_0)||duration(P_1)||duration(P_2)
  - root(P_0) is
    * $Identifier = P_0$
    * tpa(P_0)=Brake-by-Wire
    * channel(P_0)=$\perp$
    * $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
    * $initialLocation = l0$
    * $Edges = \{(l0, P_0.*.a!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l0, P_0.\mathsf{front.s}_{\mathsf{Position}}!, \emptyset, \{n++\}, \emptyset, l0),$ $(l0, P_0.\mathsf{rear.s}_{\mathsf{Position}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_0\mathsf{front.f}_{\mathsf{Position}}?, \emptyset, \{n--\}, \emptyset, l0),$ $(l0, P_0.\mathsf{rear.f}_{\mathsf{Position}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, \perp.\mathsf{f}_{\mathsf{Brake\text{-}by\text{-}Wire}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0.*.u?, n = 0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD), (l0, P_0.*.\mathsf{signal1}?, \emptyset, \emptyset, \emptyset, l0),$ $(l0, P_0.*.\mathsf{signal2}?, \emptyset, \emptyset, \emptyset, l0)\}$
  - duration(P_1) is
    * $Identifier = P_1$
    * tpa(P_1)=Position
    * channel(P_1)=P_0.front
    * *The WCET (input from the game analyzer)=Unknown*
    * $Locations = \{l_0^{P_1} : true, l_1^{P_1} : true\}$
    * $initialLocation = l_0^{P_1}$
    * $Edges = \{(l_1^{P_1}, P_0.\mathsf{front.f}_{\mathsf{Position}}!, \emptyset, \emptyset, \emptyset, l_0^{P_1}), (l_0^{P_1}, P_0.\mathsf{front.s}_{\mathsf{Position}}?, \emptyset, \emptyset, \emptyset, l_1^{P_1})\}$

- duration(P_2) is
  - $Identifier = P_2$
  - tpa(P_2)=Position
  - channel(P_2)=P_0.rear
  - *The WCET (input from the game analyzer)=Unknown*
  - $Locations = \{l_0^{P_2} : true, l_1^{P_2} : true\}$
  - $initialLocation = l_0^{P_2}$
  - $Edges = \{(l_1^{P_2}, P_0.\mathsf{rear.f_{Position}}!, \emptyset, \emptyset, \emptyset, l_0^{P_2}), (l_0^{P_2}, P_0.\mathsf{rear.s_{Position}}?, \emptyset, \emptyset, \emptyset, l_1^{P_2})\}$

Name	Type	Purpose
x	Clock	To record time passed since S was initialized
y	Clock	To record time passed since W was initialized
z	Clock	To record time passed since D was initialized
aS1	Boolean	To record whether $core_1$ is currently assigned (1) to execute S or not (0)
aW1	Boolean	To record whether $core_1$ is currently assigned (1) to execute W or not (0)
aD1	Boolean	To record whether $core_1$ is currently assigned (1) to execute D or not (0)
aS2	Boolean	To record whether $core_2$ is currently assigned (1) to execute S or not (0)
aW2	Boolean	To record whether $core_2$ is currently assigned (1) to execute W or not (0)
aD2	Boolean	To record whether $core_2$ is currently assigned (1) to execute D or not (0)
aS3	Boolean	To record whether $core_3$ is currently assigned (1) to execute S or not (0)
aW3	Boolean	To record whether $core_3$ is currently assigned (1) to execute W or not (0)
aD3	Boolean	To record whether $core_3$ is currently assigned (1) to execute D or not (0)
iS	Boolean	To record whether S is initialized (1) or yet to initialize (0)
iW	Boolean	To record whether W is initialized (1) or yet to initialize (0)
iD	Boolean	To record whether D is initialized (1) or yet to initialize (0)
uW	Boolean	To record whether an update in W is performed (1) or not (0)
uD	Boolean	To record whether an update in D is performed (1) or not (0)
L1	Integer	To record the current worst possible loads on $core_1$
L2	Integer	To record the current worst possible loads on $core_2$
L3	Integer	To record the current worst possible loads on $core_3$
vS	Integer	To record the current value in the speedometer
sD	Integer	To record the current door state
F	Integer	To record the current total number of core failures

Table 10. Variables in the monolithic model

```
void start(int[1,3] task){
if (task==S){L1=L1+S1*aS1;       L2=L2+S2*aS2;   L3=L3+S3*aS3;   iS=1;}
else if (task==W){L1=L1+W1*aW1; L2=L2+W2*aW2;   L3=L3+W3*aW3;   iW=1;}
else if (task==D){L1=L1+D1*aD1; L2=L2+D2*aD2;   L3=L3+D3*aD3;   iD:=1;}}
```

Fig. 32. Function start in the monolithic and compositional models

Name	Type	Purpose
CFL	Constant	To store CFL
pS	Constant	To store release period of S
pW	Constant	To store release period of W
pD	Constant	To store release period of D
wS	Constant	To store the WCET of S
wW	Constant	To store the WCET of W
wD	Constant	To store the WCET of D
bS	Constant	To store the BCET of S
bW	Constant	To store the BCET of W
bD	Constant	To store the BCET of D
lS1	Constant	To store the worst-case load of S on $core_1$
lS2	Constant	To store the worst-case load of S on $core_2$
lS3	Constant	To store the worst-case load of S on $core_3$
lW1	Constant	To store the worst-case load of W on $core_1$
lW2	Constant	To store the worst-case load of W on $core_2$
lW3	Constant	To store the worst-case load of W on $core_3$
lD1	Constant	To store the worst-case load of D on $core_1$
lD2	Constant	To store the worst-case load of D on $core_2$
lD3	Constant	To store the worst-case load of D on $core_3$
Limit	Constant	To store the load limit of every core

Table 11. Constants in the monolithic model

Name	From	To	Purpose
sS	service	S	To start S
sW	service	W	To start W
sD	service	D	To start D
fS	service	S	To finish S
fW	service	W	To finish W
fD	service	D	To finish D

Table 12. Actions in the monolithic model

```
void finish(int[1,3] task){
if (task==S){L1=L1-S1*aS1;      L2=L2-S2*aS2;   L3=L3-S3*aS3;   iS=0;}
else if (task==W){L1=L1-W1*aW1; L2=L2-W2*aW2;   L3=L3-W3*aW3;   iW=0;}
else if (task==D){L1=L1-D1*aD1; L2=L2-D2*aD2;   L3=L3-D3*aD3;   iD=0;}}
```

Fig. 33. Function `finish` in the monolithic and compositional models

Name	Type	Purpose
x	Clock	To record time passed since S was initialized
y	Clock	To record time passed since W was initialized
z	Clock	To record time passed since D was initialized
aS1	Boolean	To record whether $core_1$ is currently assigned (1) to execute S or not (0)
aW1	Boolean	To record whether $core_1$ is currently assigned (1) to execute W or not (0)
aD1	Boolean	To record whether $core_1$ is currently assigned (1) to execute D or not (0)
aS2	Boolean	To record whether $core_2$ is currently assigned (1) to execute S or not (0)
aW2	Boolean	To record whether $core_2$ is currently assigned (1) to execute W or not (0)
aD2	Boolean	To record whether $core_2$ is currently assigned (1) to execute D or not (0)
aS3	Boolean	To record whether $core_3$ is currently assigned (1) to execute S or not (0)
aW3	Boolean	To record whether $core_3$ is currently assigned (1) to execute W or not (0)
aD3	Boolean	To record whether $core_3$ is currently assigned (1) to execute D or not (0)
iS	Boolean	To record whether S is initialized (1) or yet to initialize (0)
iW	Boolean	To record whether W is initialized (1) or yet to initialize (0)
iD	Boolean	To record whether D is initialized (1) or yet to initialize (0)
L1	Integer	To record the current worst possible loads on $core_1$
L2	Integer	To record the current worst possible loads on $core_2$
L3	Integer	To record the current worst possible loads on $core_3$
F	Integer	To record the current total number of core failures

Table 13. Variables in the compositional model

Name	Type	Purpose
CFL	Constant	To store CFL
pS	Constant	To store release period of S
pW	Constant	To store release period of W
pD	Constant	To store release period of D
wS	Constant	To store the WCET of S
wW	Constant	To store the WCET of W
wD	Constant	To store the WCET of D
bS	Constant	To store the BCET of S
bW	Constant	To store the BCET of W
bD	Constant	To store the BCET of D
lS1	Constant	To store the worst-case load of S on $core_1$
lS2	Constant	To store the worst-case load of S on $core_2$
lS3	Constant	To store the worst-case load of S on $core_3$
lW1	Constant	To store the worst-case load of W on $core_1$
lW2	Constant	To store the worst-case load of W on $core_2$
lW3	Constant	To store the worst-case load of W on $core_3$
lD1	Constant	To store the worst-case load of D on $core_1$
lD2	Constant	To store the worst-case load of D on $core_2$
lD3	Constant	To store the worst-case load of D on $core_3$
Limit	Constant	To store the load limit of every core

Table 14. Constants in the compositional model

```
void reassign(int[1,3] task, int[1,3] from, int[1,3] to){
if (task==S && from==2 && to==1){aS2=0; aS3=0;  aS1=1;  L2=L2-iS*S2;    L1=iS*S1;       F--;}
else if(task==S && from==3 && to==1){aS2=0;     aS3=0;  aS1=1;  L3=L3-iS*S3;    L1=iS*S1;       F--;}
else if(task==W && from==1 && to==2){aW2=1;     aW3=0;  aW1=0;  L1=L1-iW*W1;    L2=iW*W2;       F--;}
else if(task==W && from==3 && to==2){aW2=1;     aW3=0;  aW1=0;  L3=L3-iW*W3;    L2=iW*W2;       F--;}
else if(task==D && from==1 && to==3){aD3=1;     aD2=0;  aD1=0;  L1=L1-iD*D1;    L3=iD*D3;       F--;}
else if(task==D && from==2 && to==3){aD3=1;     aD2=0;  aD1=0;  L2=L2-iD*D2;    L3=iD*D3;       F--;}
else if (task==S && from==1 && to==2){aS2=1;    aS1=0;  aS3=0;  L2=L2+iS*S2;}
else if (task==S && from==1 && to==3){aS2=0;    aS1=0;  aS3=1;  L3=L3+iS*S3;}
else if (task==W && from==2 && to==1){aW1=1;    aW2=0;  aW3=0;  L1=L1+iW*W1;}
else if (task==W && from==2 && to==3){aW1=0;    aW2=0;  aW3=1;  L3=L3+iW*W3;}
else if (task==D && from==3 && to==1){aD2=0;    aD1=1;  aD3=0;  L1=L1+iD*D1;}
else if (task==D && from==3 && to==2){aD2=1;    aD1=0;  aD3=0;  L2=L2+iD*D2;}
else if (task==S && from==2 && to==3){aS1=0;    aS2=0;  aS3=1;  L3=L3+iS*S3;}
else if (task==S && from==3 && to==2){aS1=0;    aS2=1;  aS3=0;  L2=L2+iS*S2;}
else if (task==W && from==1 && to==3){aW1=0;    aW2=0;  aW3=1;  L3=L3+iW*W3;}
else if (task==W && from==3 && to==1){aW1=1;    aW2=0;  aW3=0;  L1=L1+iW*W1;}
else if (task==D && from==1 && to==2){aD2=1;    aD1=0;  aD3=0;  L2=L2+iD*D2;}
else if (task==D && from==2 && to==1){aD1=1;    aD2=0;  aD3=0;  L1=L1+iD*D1;}}
```

Fig. 34. Function `reassign` in the monolithic and compositional models

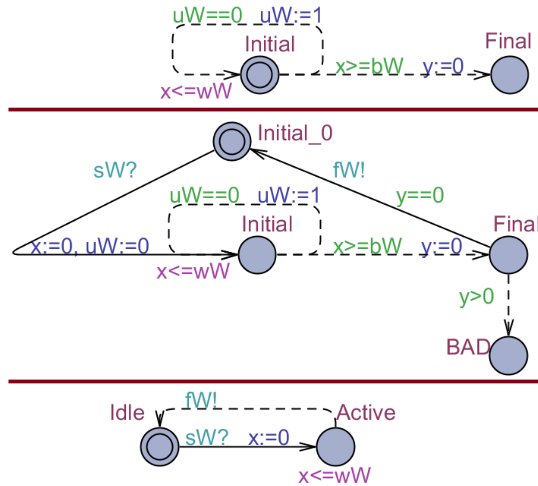


Fig. 35. Timed process automaton (in the top), standalone automaton (in the middle), and duration automaton (in the bottom) of task `W` of Section 2

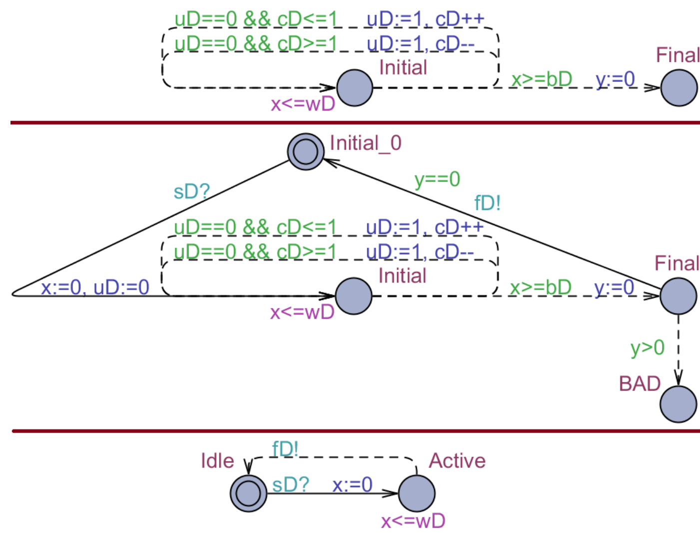


Fig. 36. Timed process automaton (in the top), standalone automaton (in the middle), and duration automaton (in the bottom) of task `D` of Section 2