Controller Synthesis for Dynamic Hierarchical Real-Time Plants Using Timed Automata

Md Tawhid Bin Waez[1], Andrzej Wąsowski[2], Juergen Dingel[3], and Karen Rudie[3]

[1] Ford Motor Company, USA mwaez@ford.com
[2] IT University of Copenhagen, Denmark wasowski@itu.dk
[3] Queen's University, Canada {dingel@cs,karen.rudie@}.queensu.ca

Abstract. We use timed I/O automata based timed games to synthesize task-level reconfiguration services for cost-effective fault tolerance in a case study. The case study shows that state-space explosion is a severe problem for timed games. By applying suitable abstractions, we dramatically improve the scalability. However, timed I/O automata do not facilitate algorithmic abstraction generation techniques. The case study motivates the development of timed process automata to improve modeling and analysis for controller synthesis of time-critical plants which can be hierarchical and dynamic. The model offers two essential features for industrial systems: (i) compositional modeling with reusable designs for different contexts, and (ii) state-space reduction technique. Timed process automata model dynamic networks of continuous-time communicating plant processes which can activate other plant processes. We show how to establish safety and reachability properties of timed process automata by reduction to solving timed games. To mitigate the state-space explosion problem, an algorithmic state-space reduction technique using compositional reasoning and aggressive abstractions is also proposed. In this article, we demonstrate the theoretical framework of timed process automata and the effectiveness of the proposed state-space reduction technique by extending the case study.

1 Introduction

Technology giants and new automotive manufacturing companies have imposed pressure on major automotive manufacturers to be utmost cost- and time-competitive while delivering new levels of autonomous vehicles, such as driver-in-the-loop autonomous cars, or fully autonomous cars. Autonomous vehicles development has brought several system-level real-time control problems that need to be solved with formal guarantees. The new autonomous features—such as adaptive cruise control, lane keeping, and super cruise—make automotive systems too complex to manually synthesize correct controllers. In practice, MIL, SIL, PIL, HIL and vehicle-level testings are performed to test these manual synthesized controllers, which is an extremely time-consuming and expensive verification process but still cannot make formal guarantees with respect to the requirements. We present a reconfiguration controller synthesis problem: to synthesize a

system-level real-time controller with formal guarantees [1]. Many autonomous car related logic-level control problems (or controller synthesis problems) could be reduced to this presented problem. Classical control theory is not a natural solution for system-level (or logic-level) control problems, which put automotive manufacturers in an unfamiliar situation because (almost) all of their control engineers are only familiar with classical control. Correct-by-construction controller synthesis for hybrid plants can be applied to solve these control problems. Hybrid controller synthesis is performed in three major steps: i) converting continuous dynamics of the plant into an abstract discrete model, ii) synthesizing a controller for the discrete model, and iii) converting the controller of the discrete model into the controller for the actual model. The third step is the lowest computational step while the second step is too expensive to apply for industrial problems. Unfortunately, the first step is computationally the most expensive, and suffers state-space explosion even for toy problems. For these reasons, in this paper, we consider a *timed automata* (TA)-based [2,3,4] controller synthesis as TA are the simplest kind of hybrid automata with a continuous time domain [4,5].

An *open(-loop) plant* continuously interacts with an unpredictable environment. A *hierarchical plant* is a hierarchical composition of smaller plants. A *dynamic hierarchical plant* is a hierarchical plant whose components may change over time. Dynamic hierarchical 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 a case study to construct a fault-tolerant framework for a hierarchical open plant 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 plant exhibits dynamic hierarchical plant behaviors [1]. Models of industrial dynamic hierarchical open plants can be very detailed because of the hierarchical composition. These details may introduce errors in the design and make modeling and analysis challenging.

Automata are a prominent group of models in model-based development because they facilitate many important types of formal analyses. *Finite automata* 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 the exact sequence of actions but also on 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 TA, which can be viewed as finite automata with continuous clocks to record time. Real-time reachability and some other important analytical properties

of real-time formal models (such as TA, timed Petri nets [6], timed transition systems [7], and Modecharts [8]) were first solved using symbolic semantics *region graph* of TA and after that other models adopted that approach.

Timed automata are desirable for the modeling of *open real-time plants* 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 plants. This absence may lead to cumbersome design details in a large-scale plant having several *control hierarchies*. *Timed game automata* [9,10,11]—a variant of TA—are a well-known model in the research community for the controller synthesis of dense-time plants. Dense-time formal methods of TA may provide the most accurate analysis, however TA, currently, are not suited for open plants in practice mainly because of poor scalability. We propose timed process automata (TPA), a variant of TA, together with a state-space reduction technique for the compositional hierarchical modeling and controller synthesis of dynamic hierarchical open real-time plants [12]. *The case study of the task-level reconfiguration technique* of [1] could be considered as the main motivation for the development of TPA [12]. With *a case study of TPA*, this paper demonstrates (i) the theoretical framework of [12] and (ii) the effectiveness of the proposed state-space reduction technique of [12]. The case study of this paper is reproduced from the case study of [1] by applying TPA along with timed I/O automata (TIOA) [13,14,11,3].

1.1 Goals

The first goal of this paper is to develop a 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 plants, where a designer will not need to readjust a design for different compositions. However, the main motivation is to develop a state-space reduction technique for TA-based controller synthesis of dynamic hierarchical open plants.

Reconfiguration Service Synthesis Multi-core systems may 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 a formal guarantee that the developed technique can handle all fault scenarios. Automated formal synthesis of such reconfiguration frameworks is highly desirable for industrial use.

Compositional Modeling with Reuse Figure 1 presents an abstract Brake-by-Wire system modeled using TIOA. 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

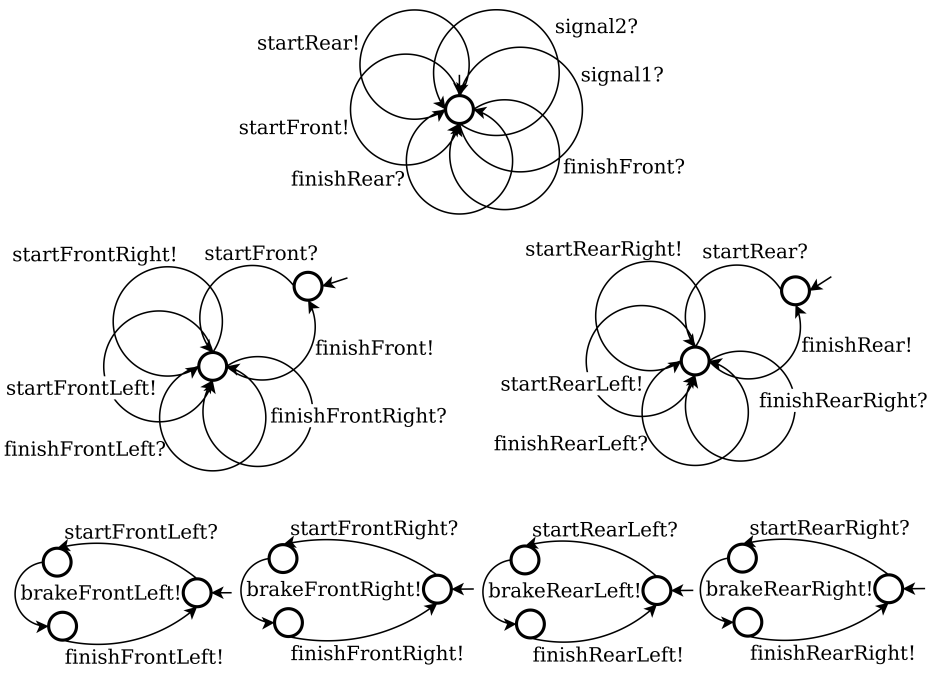

Fig. 1. An abstract Brake-by-Wire system modeled using standard TIOA, where one copy of the *main thread* of Brake-by-Wire (in the top), two copies of the main thread of Position (in the middle), and four copies of Actuator system (in the bottom)

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. These automata communicate among themselves by exchanging inputs and outputs. For example, the main thread of Brake-by-Wire receives inputs (signal1 and signal2) from the environment, receives inputs (finishFront and finishRear) from its children, and sends outputs (startFront and startRear) to its 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 would 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 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. The same Brake-by-Wire system is modeled in Section 3.1 by using 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.

State-Space Reduction Technique No algorithmic state-space reduction technique has been developed for the controller synthesis of dynamic hierarchical open dense-time plants. During our two case studies, we noticed that even a (practically) very small real-time plant may have a state space too large for automated formal controller synthesis because of hierarchy, dynamic behaviors, and time calculations [12]. We overcame the scalability problem in one of the projects—construction of a fault-tolerance framework in Section 2 and in [1]—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 plant is infeasible. A generalized algorithmic reduction technique, therefore, is needed for controller synthesis of dynamic hierarchical open time-critical plants, which is provided by presenting a reduction technique for TPA.

1.2 Problem

The problem we address is to develop a synthesis technique for reconfiguration services using TA, to develop a theoretical foundation for TA to allow compositional modeling with reuse for dynamic hierarchical open plants, and to allow timed games-based controller synthesis for larger dynamic hierarchical open plants.

Challenges The main challenge to develop service-based solution for task-level reconfigurations to achieve fault tolerance for mixed-criticality multi-core systems is to provide a formal guarantee. This paper synthesizes these services with formal assurance by reducing the problem into a safety game of TIOA. Analyses of timed games have extremely poor scalability, and no efficient state-space reduction technique is known for TIOA. 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 plants, unfortunately, typically are very detailed. For reuse in compositional modeling, we need algorithmic 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.

Methodology Timed I/O automata model and analyze a hierarchical open real-time plant by using a parallel composition of all components. Parallel composition increases the state space exponentially thus leading to state-space explosion

very quickly. We present the case study—service-based solution for task-level reconfigurations—in Section 2 to show this phenomenon. Experimental results of this case study hint that reducing the number of components in parallel composition and increasing abstraction of the components reduces the state space. However, there is no algorithmic way to reduce the number of components and increase the abstractions of the components in TIOA. Section 3 presents TPA to improve the scalability of TIOA for analyzing hierarchical open real-time plants by utilizing the lessons learned from the case study of the previous section. 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 algorithmic analysis of larger dynamic hierarchical open plants. Timed process automata allow algorithmic controller synthesis for safety and reachability properties of arbitrary number of processes; but there is an implicit bound on the maximal number of active processes at a time. Timed process automata also model and analyze using parallel composition. However, the analyzer may avoid parallel composition of all components during analysis of the plant by creating multiple parallel compositions of smaller number components, where most of the components are coarse abstractions of the original components: the analyzer may perform analysis in multiple bottom-up steps by first analyzing leaf components one by one, then using coarse abstractions of the analyzed components to analyze the next level components, and repeating the second step until the whole plant is analyzed. The paper is organized as follows:

Section 2 A 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 plants.
Section 3.2 A controllability analysis technique for the developed model.
Section 3.3 A state-space reduction technique to analyze larger dynamic hierarchical open plants.
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.
Section 4 Concludes the paper, classifies TPA, and suggests future work.

1.3 Background

A *ground hierarchical open plant* is a hierarchical (open) plant that does not have a component. A non-ground hierarchical open plant is a *compound hierarchical open plant*. A ground hierarchical open plant has a control hierarchy of depth 0. A compound hierarchical open plant 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.

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. To be in a location, the clock valuation has to satisfy the local invariant of that location. Local invariants are used to ensure the progress of the model [15], that is, control can stay in a location until its local invariant is satisfied. 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* [16,11,3]. A *timed I/O automaton* [13,14,11] 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 *[16,11,3] 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*, nonnegative 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 *[13,14,11,3] 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 *[14,11] 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 [9,17,18] between the controller and the environment, where the controller aims to find a strategy to guarantee that the plant will satisfy a given property, no matter what the environment does [19]. An example of such reformulation is to find a strategy for the controller (or a reconfiguration service) to prevent the plant from becoming unstable in the presence of the faults of the fault model. The *game reachability* problem is whether the plant has a strategy or controller 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 [20], a TIOA-based tool, is a tool for solving games based on timed game automata with respect to reachability and safety properties. Synthia [21] performs verification and controller synthesis for timed games.

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) [1]. 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. Our timed games-based approach guarantees fault-tolerance up to a certain maximal number of concurrent faults after inserting the services into the plant. Such reliable and accurate information is helpful to build mixed-criticality systems cost effectively. We demonstrate the synthesis process using a small plant, which is complex enough to show the essence of the problem and our approach, yet simple enough to allow a compact and comprehensible presentation.

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 plants 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*
- $\texttt{load}(\mathsf{task}_i, \mathsf{core}_j)$ *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* $\texttt{primary}(\mathsf{task}_i)$ *maps* task_i *to the core on which the task runs in the initial system-state*
- *Predicate* $\texttt{critical}(\mathsf{task}_i)$ *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[4]*. 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 $\mathtt{S}$, $\mathtt{W}$, $\mathtt{D}$, $\mathtt{N}_1$, $\mathtt{N}_2$, and $\mathtt{N}_3$. Only three tasks $\mathtt{S}$, $\mathtt{W}$, and $\mathtt{D}$ are the critical tasks, where in an execution $\mathtt{S}$ records exactly one update of a speedometer, and $\mathtt{W}$ (respectively, $\mathtt{D}$) 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.

Figure 2 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 $\mathtt{S}$, and $\mathtt{S}$ is a primary task of core_1 in Figure 2. 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 2, core

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

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. 2. Sample trace of $system_1$ with reconfiguration

$core_2$ 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 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 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, non-critical task 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 $service_i$ for each core $core_i$ such that $service_i$: (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 $service_i$ 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 $service_i$ executes. All

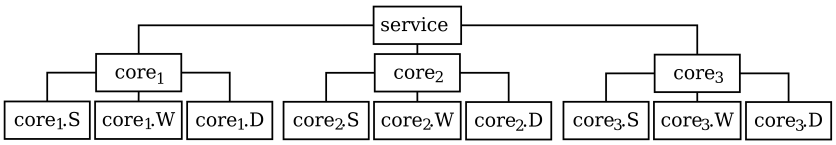

Fig. 3. Architecture of system_1 after adapting abstractions of Section 2.3

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 plant 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. Assumptions (i) and (iii) do not prevent synthesis of a distributed reconfiguration service per core, which will be shown in Section 2.4. Assumption (ii) considers the worst-case core load by which no possible total core load can force the synthesized controller to fail. 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`).

First we construct 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 [11]. 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 3. 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 $\mathsf{core}_1.\mathtt{S}$ represents the

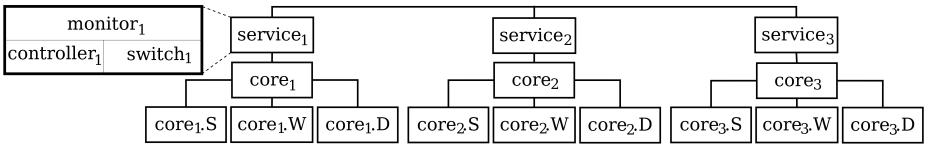

Fig. 4. Architecture of system_1 at runtime

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 represents the activities of core core_1. The `service` automaton 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 a technical report [22].

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 plant 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 4 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 unsafe locations. If the property holds, the strategy—which is our central controller—is 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[5] 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 2 the controller has non-deterministic choices at system-state s_4 when only core_2 fails and the other two cores are operational. 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 2) and some paths do not (say reallocating W to core_1). For example, when core_2 fails 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 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 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 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.

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

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. The abstract model has only one automaton, which is presented in [22].

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. 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 of this paper 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

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

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 observations 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 BCET 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 2 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.

3 Timed Process Automata

We propose *timed process automata (TPA)*, a variant of TA, for the *synthesis of controllers* for *safety* and *reachability* properties in *time-critical plants* [12]. To fulfill industrial requirements, we consider time-critical plants that are open (communicate with external components without a controller), hierarchical (can be decomposed and recomposed into smaller plants), and dynamic (the decomposition can change over time). The proposed variant provides compositional modeling with reuse for three different contexts and algorithmic analysis—a plant 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 plant or components of different larger plants or a combination of all previous scenarios.

3.1 Processes

This section presents the syntax and the semantics: i) TPA model processes in a way that each process is a dynamic hierarchical open time-critical plant, ii) every process hierarchically contains its active callee processes—thus the control

of a process is hierarchically shared with its active callee processes, iii) the main thread of a process can activate callee processes via communication channels, iv) an active process can receive any input in any state, v) an active callee process can deactivate itself in any state of the main thread of its caller process, and vi) an activated callee process terminates within its worst-case execution time.

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* 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 5 presents TPA Brake-by-Wire, Position, and Actuator. They represent the same Brake-by-Wire system of Figure 1. In Figure 5, each initial location

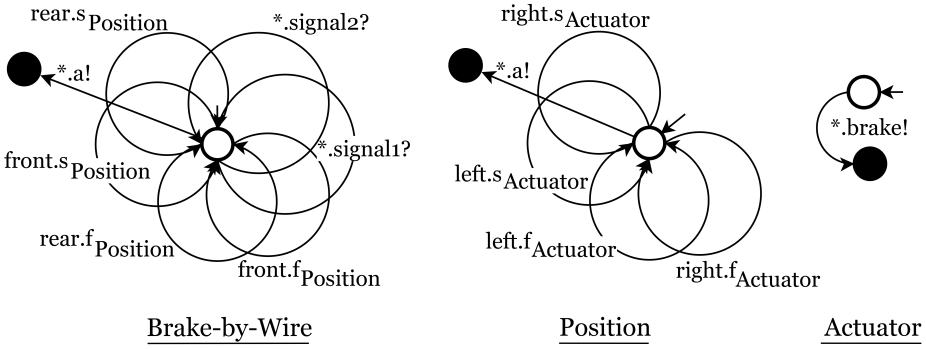

Fig. 5. The same Brake-by-Wire system of Figure 1 is modeled using TPA

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 $\mathsf{id}(P)$[6] 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. The *public channel*, the only non-private channel, is used to communicate with the environment. 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 $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_f)$ 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 $|C| \times |A_s|$ active callee processes. For example, an instance of automaton Brake-by-Wire of Figure 5 can activate at most two instances of automaton Position of Figure 5 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 5. 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.

A process P starts a process Q of an automaton $\mathsf{tpa}(Q)$ via channel $P.c$ by traversing an edge $e_1 = (\_, \mathrm{s}_{\mathsf{tpa}(Q)}, c, \_, \_, \_)$ labeled by a start action $\mathrm{s}_{\mathsf{tpa}(Q)}$ if there exists no active process of $\mathsf{tpa}(\mathrm{Q})$ with caller channel $P.c$; dually, P traverses an edge $e_2 = (\_, \mathrm{f}_{\mathsf{tpa}(Q)}, c, \_, \_, \_)$ labeled by the paired finish action $\mathrm{f}_{\mathsf{tpa}(Q)}$ whenever Q reaches its final state. No edge labeled by $\mathrm{f}_{\mathsf{tpa}(Q)}$ 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 $\mathrm{f}_{\mathsf{tpa}(Q)}$ 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 [3,13,14,11].

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

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* $\mathcal{S}[\![P]\!] = (L \times \mathbb{R}^X_{\geq 0} \times \{P\}, (l_0, \mathbf{0}, P), A^P, \twoheadrightarrow)$[7]*, 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* $(l, v, P) \overset{d}{\twoheadrightarrow} (l, v + d, P)$.

A *standalone state* of a process P is (l, v, P), where l is a location and v is a clock valuation.

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 standalone state that can be reached from standalone state (l, v, P) after delaying $d \in \mathbb{R}$ time units is $(l, v + d, P)$,
- The only standalone state that can be reached from standalone 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 standalone state that can be reached from standalone 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 ($A_s \cup A_f = \emptyset$). Automaton Actuator in Figure 5, for instance, is a ground timed process automaton. *Compound TPA* are TPA that can perform a start or finish action ($A_s \cup A_f \neq \emptyset$). For example, Brake-by-Wire and Position in Figure 5 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

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

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 5, 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 $st^P \xrightarrow{P.c.a} st'^P$ 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:*

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

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

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

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

$$\frac{st^Q \overset{W.c.a}{\longrightarrow\!\!\!\!\!\rightarrow} st'^Q \text{ and } a \in A_{\circ}^Q \text{ and } st^Q \in s \qquad s' = \{st^R \in s \mid st^R \overset{W.c.a}{\longrightarrow\!\!\!\!\!\rightarrow} st'^R \text{ and } W.c \text{ is a channel}\}}{s \xrightarrow{Q.c.a} \{s \setminus s'\} \cup \{st'^R \mid st^R \overset{W.c.a}{\longrightarrow\!\!\!\!\!\rightarrow} st'^R \text{ and } st^R \in s\}} \textsc{Output}$$

A *global state* (s) is a set which holds standalone states of all active processes. The START rule states that the initial standalone state (st_0^R) of a freshly started callee (R) is added to the global state whenever the corresponding start action ($Q.c.\mathrm{s}_T$) is performed by its caller (Q). The rule also states that no two active processes can have the same timed process automaton ($\mathsf{tpa}(W) = T$) and the same caller channel ($\mathsf{channel}(W) = Q.c$). The FINISH rule prescribes that the standalone-final state (st_{f}^R) of a callee (R) is removed from the global state and the caller executes the corresponding finish action ($Q.c.\mathrm{f}_{\mathsf{tpa}(R)}$) whenever that callee is in the standalone-final state and no standalone state (st^U) 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 (Q) receives an input (a) from the environment via channel $id.*$ ($Q.*$). Rule OUTPUT declares a process (Q) send an output (a) via channel $id.c$ ($W.c$)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 global state that can be reached from global 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 global state that can be reached from global 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 global state that can be reached from global 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 controller synthesis of *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 plant 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 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 plant 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

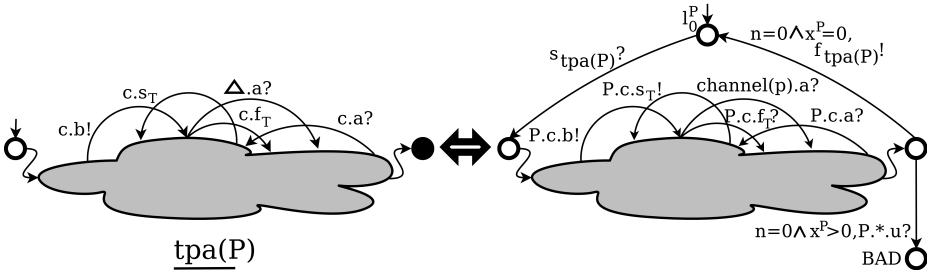

Fig. 6. A generalized view of the standalone automata construction

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[8] 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 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 $\triangle$:*

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

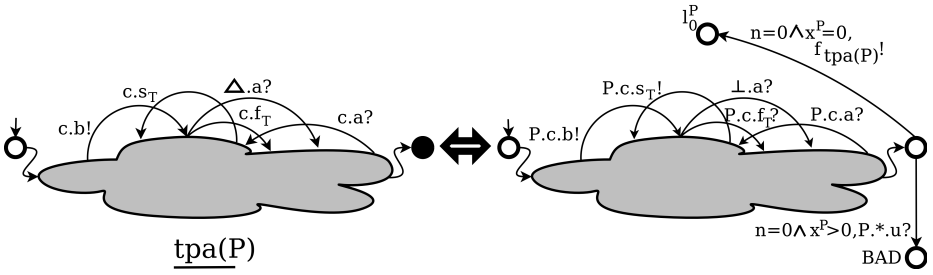

Fig. 7. A generalized view of the root automata construction

- *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 algorithmic process.

Definition 10 *The monolithic analysis model of a ground timed processes automaton (such as* $\mathsf{Actuator}$*) 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* $(\_, \mathrm{s}_T, c, \_, \_, \_) \in E$*.*

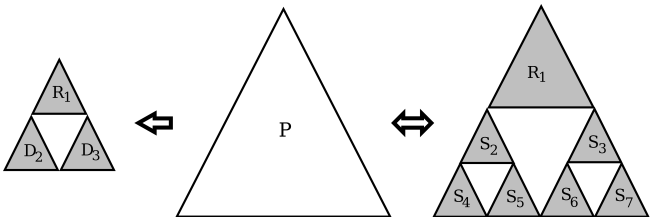

Fig. 8. A compositional (sound) analysis model on the left and a monolithic (sound and complete) analysis model on the right of automaton Brake-by-Wire, where P is a process of the automaton, R_1 is the root automaton, S_2–S_7 are standalone automata, and D_2–D_3 are duration automata

Figures 6–7 present a generalized view of the standalone and root automata constructions ([22] presents 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 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 Algorithmic State-Space Reduction

We introduce a 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 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 8)—so called *duration automata*.

Duration Automata A duration automaton (Figure 9) 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* [23,24] for which the controller has a winning strategy to reach that target location by avoiding unsafe states. Like [25,26], 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 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 10 presents the compositional analysis procedure of Brake-by-Wire (the detailed models are presented in [22]). The compositional model construction

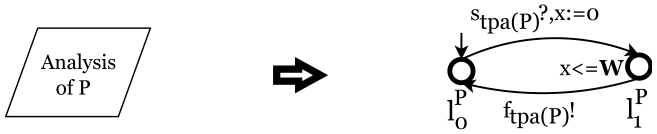

Fig. 9. A generalized view of duration automata construction

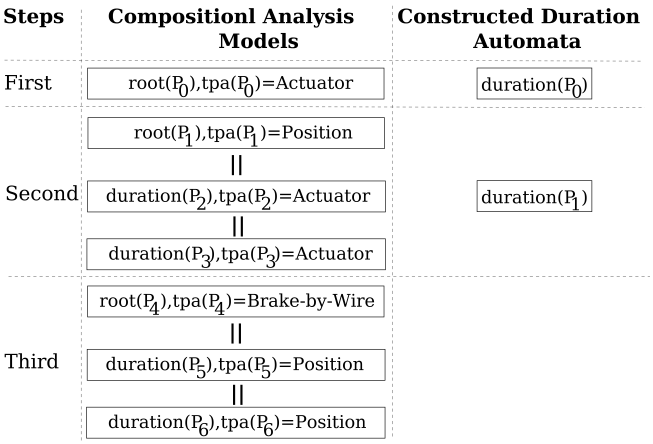

Fig. 10. Steps of the compositional analysis of automaton Brake-by-Wire: root(P_0), tpa(P_0) = Actuator means root automaton of process P_0, where P_0 is an instance of Actuator, and similar interpretations apply for others.

procedure terminates, and the composition of all the above TIOA is a TIOA, because we consider only non-recursive well-formed TPA.

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 algorithmic 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 10, 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 [22]. 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).

All automata of this experiment are presented in [22]. First, we model the central reconfiguration service and three tasks: `S`, `W`, and `D` using TPA. The service automaton models task releases and terminations. An unsafe location in this automaton is unreachable—a central reconfiguration service (or a controller) exists that makes the plant 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 service automaton. 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 of the TPA representing tasks and the root automaton of the timed process automaton representing the central reconfiguration service. 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. 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. We performed the same experiments on the compositional model and the and the monolithic model that we performed on the concrete model (in Section 2), and the abstract

model (in Section 2). Table 2 shows that the compositional model produces a much smaller state space than the monolithic model.

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). 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. 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 BCET 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.

Observations in the above match with the observations presented in Section 2.5. The concrete model and the monolithic model could be constructed using different abstractions or assumptions; consequently, a comparison between these two models may convey misleading insights. Similarly, comparing the results of the abstract model and the compositional model may not be useful. The main difference

Configurations of Table 1	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 2. Comparisons of the concrete, abstract, monolithic and compositional models with respect to synthesis time (in seconds) and the strategy size (in kilobytes)

between these two models is the algorithmic construction (of the compositional model), which is a key requirement for industrial usage.

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 plant 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 systems. Our analysis techniques can be applied in practice using any standard timed games solver such as Uppaal Tiga [20] and Synthia [21]. Timed process automata can model private communication and open systems. Moreover, TPA provide two important features for dynamic open time-critical systems development: (i) compositional modeling with reusable designs for different contexts and (ii) algorithmic state-space reduction technique.

Classification

Timed process automata fall in the class of *TA with resources* [4] 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. Task automata [27,4] 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. More precisely, the model is a direct generalization of *task automata* [28], *dynamic networks of TA* [29], and *callable TA* [30]. 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 [29], 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 algorithmic state-space reduction technique.

Dynamic networks of continuous-time automata have also been studied in the context of hybrid automata [31,32]. 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 [32]. 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* [4] because they hide many design details from the designers to achieve succinctness (like *TA variants with urgency* [33,34,4]). Timed process automata are also timed game automata [9,18,14,11] because the new variant uses timed games for analysis.

Limitations and Future Work

Use of TA in 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 [35]. 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: to improve computational efficiency of 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 [22]. We, however, do not know other design aspects for which manual design alterations can be replaced by algorithmic 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 algorithmic state-space reduction technique for timed games-based controller synthesis of dynamic hierarchical plants. Theoretically manual state-space reduction may achieve similar or smaller state-spaces than algorithmic state-space reduction. Even practically it is usually true for smaller systems for example, the comparisons in Table 2. However, efficiency of algorithmic 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, algorithmic state-space reduction techniques for standard TIOA are much more important and desirable than algorithmic state-space reduction techniques for TPA. We strongly

encourage researchers to develop an algorithmic state-space reduction technique for standard TIOA. A similar but larger challenge is to develop a 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 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 [36,37].

It would be interesting to consider a model transformation from a subset of the *real-time π-calculus* [38,39] 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 [38], 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* [40,41].

References

1. 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
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. 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
5. Alur, R., Dill, D.L.: Automata-theoretic verification of real-time systems. In: Formal Methods for Real-Time Computing. Trends in Software Series. John Wiley & Sons Publishers (1996) 55–82
6. Ramchandani, C.: Analysis of asynchronous concurrent systems by timed Petri nets. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1974)
7. Ostroff, J.S.: Temporal Logic for Real Time Systems. John Wiley & Sons, Inc., New York, NY, USA (1989)

8. Jahanian, F., Mok, A.K.: Modechart: A specification language for real-time systems. IEEE Transactions On Software Engineering **20**(12) (December 1994) 933–947
9. 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
10. Henzinger, T.A., Kopke, P.W.: Discrete-time control for rectangular hybrid automata. Theoretical Computer Science **221** (June 1999) 369–392
11. 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
12. 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
13. 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)
14. 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
15. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and Computation **111** (1994) 394–406
16. 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
17. 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
18. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: The element of surprise in timed games. In: CONCUR. Volume 2761 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2003) 144–158
19. 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
20. 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
21. 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
22. Waez, M.T.B., Wąsowski, A., Dingel, J., Rudie, K.: Timed automata to synthesize controllers of dynamic hierarchical real-time plants. Technical Report 2016-631, Queen's University, ON (August 2016) http://research.cs.queensu.ca/TechReports/Reports/2016-631.pdf.

23. 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
24. 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
25. 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
26. 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
27. 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
28. 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
29. 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
30. 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
31. 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
32. 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)
33. 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
34. Barbuti, R., Tesei, L.: Timed automata with urgent transitions. Acta Informatica **40** (March 2004) 317–347
35. 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
36. Lawford, M.: Model Reduction of Discrete Real-Time Systems. PhD thesis, Department of Electrical Computer Engineering, University of Toronto, Toronto, ON, Canada (1997)

37. 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
38. 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
39. 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
40. 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
41. 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