Technical Report 2014-619

Title: Synthesis of a Reconfiguration Service for Mixed-Criticality Multi-Core Systems

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

School of Computing
Queen's University
Kingston, Ontario, K7L 3N6
28th April 2014

Synthesis of a Reconfiguration Service for Mixed-Criticality Multi-Core Systems

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

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

Abstract. Task-level reconfiguration techniques in automotive applications aim to reallocate tasks to computation cores during failures to guarantee that the desired functionality is still delivered. We consider a class of mixed-criticality asymmetric multi-core systems inspired by our collaboration with General Motors, for which we automatically synthesize task-level reconfiguration services to reduce the number of processing cores and decrease cost without weakening fault-tolerance. We admit the following types of faults: safety violations by tasks, permanent core failures, and temporary core failures. We use timed games to synthesize the controllers. The services suspend and reinstate the periodic executions of the non-critical tasks to ensure enough processing capacity for the critical tasks by maintaining lookup tables, which keep track of processing capacity. We present a methodology to synthesize the services and use a case study to show that suitable abstractions can dramatically improve the scalability of timed games-based tools for solving industrial problems.

1 Introduction

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

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

criticality, one of its prospective application sectors is next-generation automotive systems, most of which are expected to be mixed-criticality multi-core systems.

Formal methods have been used for the development of fault-tolerant real-time systems. However, in the automotive and avionics industries, fault-tolerance problems are typically designed, analyzed, and solved using classical control theory. Timed game theory [1,2,3], a dense-time automata-based game theory, is almost unexplored in the industry. The use of timed game theory to solve industrial problems is attractive because of automated controller synthesis, visual modeling, and dense-time formal analysis. Nevertheless, applying timed game theory to solve industrial problems is challenging because of its high computational complexity.

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

Methodology In Sect. 3 we propose a service-based task-level reconfiguration technique to guarantee fault-tolerance of mixed-criticality multi-core systems. Timed games are used to synthesize a controller that selects safe operational cores to reallocate the periodic executions of critical tasks from failed cores. Lookup tables are used at runtime to suspend and reinstate the periodic executions of non-critical tasks to ensure that operational cores have enough free processing capacity for the executions of the newly reallocated critical tasks. We synthesize the reconfiguration services in six steps:

Section 4 Identification of modeling abstractions and required system parameters to construct a scalable model.

Sections 4.1–4.3 Construction of a timed game model where unsafe locations are reachable if and only if a core exceeds its load capacity.

Section 5.1 Analysis of the model for the existence of a controller that ensures no unsafe location is reachable; binary search for the maximal value of the concurrent-failures–limit for which such a controller exists; and automated synthesis of the controller of the maximal concurrent-failures–limit.

Section 5.2 Synthesis of the services by distributing the synthesized controller and combining the abstracted elements of the first step.

Section 5.3 Leverage scalability of the whole process for industrial systems.

Section 5.4 Generalization of the synthesis process to apply in other multi-core systems, such as for symmetric multi-core processing (SMP) systems

We use Uppaal Tiga [4]—a solver for timed games—to model, analyze, and synthesize. The methodology, however, can be applied using any solvers for timed games, such as Synthia [5]. The paper concludes in Sect. 6.

2 Related Work

Timed automata (TA) [6,7]—finite automata with dense-time clocks, clock constraints, and clock resets—are a prominent class of formal models to analyze safety and reachability properties of real-time systems. Clocks, clock constraints, and clock resets are used to express timing behaviors in TA. Timed automata have been used many types of analysis purposes including for fault diagnosis [8,9,10], analyzing multi-core systems [11,12], and analyzing mixed-criticality systems [13]. However, classical TA cannot be used for controller synthesis [7].

A real-time control problem can be viewed as a two-player timed game [1,2,3] between the controller and the environment, where the controller aims to find a strategy to guarantee that the system will satisfy a given property, no matter what the environment does [14]. An example of such reformulation is to find a strategy for the controller (or the reconfiguration service) to prevent the system from becoming unstable in the presence of the faults of the fault model. To our best knowledge, no prior work considered timed games to synthesize controllers for automotive mixed-criticality fault-tolerant multi-core systems.

A timed I/O automaton [15,16] is a timed automaton which has an input alphabet and a set of uncontrollable transitions along with a regular (output) alphabet and a set of regular (controllable) transitions. The controller plays controllable transitions and the environment plays uncontrollable transitions; thus timed I/O automata (TIOA) are a natural model for real-time games. Uppaal Tiga [4] is a well-known timed games-based tool that uses TIOA for modeling.

3 Task-Level Reconfiguration Technique

We introduce a service-based task-level reconfiguration technique to assure fault-tolerance of mixed-criticality multi-core systems.

3.1 Systems

We consider a class of multi-core systems with multiple sets of symmetric processing cores, such that any two cores from two different sets are asymmetric. Symmetric cores are able to execute a task with the same performance. Asymmetric cores might not execute the same set of tasks, and different asymmetric cores may exhibit different performance for the same task. The systems under consideration are mixed-criticality systems, because they execute both critical tasks and non-critical tasks. Every task can be killed (and resumed) in any of its states by a reconfiguration technique. Tasks may invoke subtasks in a recursive fashion. The system has a fault-intolerant criticality-unaware AMP scheduler with a static allocation of tasks.

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,

developed using formal methods, are assumed not to breach any safety constraints. On the contrary, a non-critical task may violate safety constraints. Every core of the system may fail. However, all cores of a system concurrently cannot be in their failed states. The maximal number of cores that can fail concurrently is restricted by a limit, concurrent-failures–limit (CFL). Whereas no limit is imposed on the total number of fault occurrences in a run.

Problem Statement Synthesize a task-level reconfiguration framework to assure uninterrupted executions of all the critical tasks and to guard against the safety violations of all the non-critical tasks in the presence of faults. An additional, nonetheless important, requirement is determining the maximal concurrent failures for all combinations that the synthesized framework can manage.

3.2 Task-Level Reconfiguration Service

We propose a service-based reconfiguration technique to solve the problem, where the system has a task-level reconfiguration service for per core. The services manage critical tasks differently than non-critical tasks. Consider, for instance, a simple mixed-criticality AMP system system_1, one of the systems that are described in Sec. 3.1. System system_1 executes six periodic tasks `S`, `W`, `D`, $\mathtt{N}_1$, $\mathtt{N}_2$, and $\mathtt{N}_3$. Only three tasks `S`, `W`, and `D` are the critical tasks, where in an execution `S` records exactly one update of a speedometer and `W` (respectively, `D`) records at most one update of a wiper (resp., door). The system has three cores core_1, core_2, and core_3, which are asymmetric but each core is able to execute all six tasks.

Figure 1 presents a trace of a desirable behavior of system_1 in the presence of different faults after inserting the reconfiguration services; the figure omits suspended non-critical tasks to avoid clutter. At any given time, the periodic execution of a task can be assigned to at most one operational core. A task is assigned to its primary core in the initial system-state, where a core is responsible to execute only its primary tasks. For instance, core_1 is the primary core of task `S`, and `S` is a primary task of core_1 in Fig. 1. We call a non-primary core as a backup core of a critical task when that core can execute

	core_1	core_2	core_3
s_1	core_1: operational S : primary $\mathtt{N}_1$: safe	core_2: operational W: primary $\mathtt{N}_2$: safe	core_3: operational D: primary $\mathtt{N}_3$: safe
s_2	core_1: operational S: primary $\mathtt{N}_1$: safe	core_2: failed W: primary $\mathtt{N}_2$: safe	core_3: operational D: primary $\mathtt{N}_3$: safe
s_3	core_1: operational S: primary $\mathtt{N}_1$: safe	core_2: failed	core_3: operational D: primary W: safe
s_4	core_1: operational S: primary $\mathtt{N}_1$: 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 $\mathtt{N}_2$: safe	core_3: operational D: primary $\mathtt{N}_3$: safe

Fig. 1. Sample trace of system_1 with reconfiguration

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 Fig. 1, core core_2 fails in system-state s_2; in the next system-state, the periodic execution of critical task `W` is assigned to a backup core core_3 and the periodic execution of non-critical task $\mathtt{N}_3$ is suspended temporarily on core_3 to have enough processing capacity for `W`. A critical task is allowed to execute further on a backup core only if the primary core is in a failed state. The services kill a critical task on a backup core (if that task is initialized or released) and cancels the assignment of that task on that backup core, whenever the primary core recovers from a temporary failure. As an example, core core_2 recovers from a temporary failure in system-state s_6, and after that only core_2 is assigned to perform critical task `W`. The services reinstate a suspended non-critical task as soon as enough processing capacity for that task is regained due to the recovery of a core from a temporary failure; for example, the periodic execution of non-critical task $\mathtt{N}_3$ is reinstated in system-state s_7. The services permanently suspends a non-critical task if it performs some harmful activities, such as illegal memory access. For instance, non-critical task $\mathtt{N}_1$ performs some harmful activities in system-state s_4 and the task is permanently suspended in system-state s_5.

4 Modeling

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

To obtain a scalable model: (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. None of these abstractions prevents the synthesis of one individual reconfiguration service per core as shown in Sect. 5. Our model depends on four system parameters: (i) the release period of each task (constants `pS`, `pW`, `pD`); (ii) the worst-case load of each task on each core, in percent of the processing capacity of that core (constants `lS1`, `lW1`, `lD1`, `lS2`, `lW2`, `lD2`, `lS3`, `lW3`, `lD3`); (iii) the worst-case execution time (WCET) of each task on all cores (constants `wS`, `wW`, `wD`); and (iv) the best-case execution time (BCET) of each task on all cores (constants `bS`, `bW`, `bD`).

Now we illustrate the design process by constructing a concrete model of mixed-criticality AMP system system_1. The main design principle behind this model is to describe each component of the system in detail as a timed game automaton then obtain an intuitive model by composing all the components using parallel composition [16]. The concrete model has 13 timed I/O automata, which follow five different templates. Each automaton represents exactly one rectangle

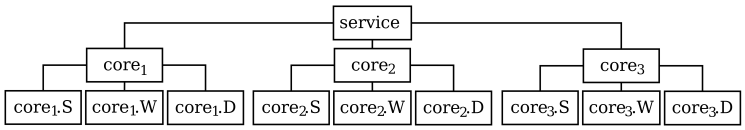

Fig. 2. Architecture of system_1 after adapting abstractions of Sect. 4

of Fig. 2. 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; for example, task automaton core_1.S in the bottom of Fig. 3 represents the activities of task S on core_1. A core may fail only if the fault model allows it to fail. A core automaton models initializations and terminations of tasks on a core along with failures of the core and safety violations; for instance, core automaton core_1 in the middle of Fig. 3 represents the activities of core core_1. The `service` automaton in the top of Fig. 3 models reallocations of the critical tasks when a core fails or recovers. In the model a failed core may recover at any time. The other ten automata of the concrete model are presented in the appendix.

The automata modeling cores follow the same template. For instance, automaton core_1 uses action `kS1` to model the killing of task S on core_1 (edge 16 in Fig. 3), `kW1` to model the killing of task W on core_1 (edge 17), `kD1` to model the killing of task D on core_1 (edge 18), and global variable `L1` to record the current worst-case load on core_1 (edges 9–14,16–18); similarly, automaton core_2 uses action `kS2` to model the killing of S on core_2 and global variable `L2` to record the current worst-case load on core_2. The automata modeling the same task—but on different cores—follow the same template.

4.1 Task Automata

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

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

clock x records the amount of time passed since the last initialization of S and global constant wS stores the WCET of S. Similarly, global constant bS stores the BCET of task S. Hence, clock guard x≥bS prevents task S to terminate before the BCET (edge 5).

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

Figures 13-16, in the appendix, present task automata core_1.W and core_1.D in the concrete model. The automata model task life-cycle activities and task specific activities of tasks W and D.

4.2 Core Automata

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

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

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

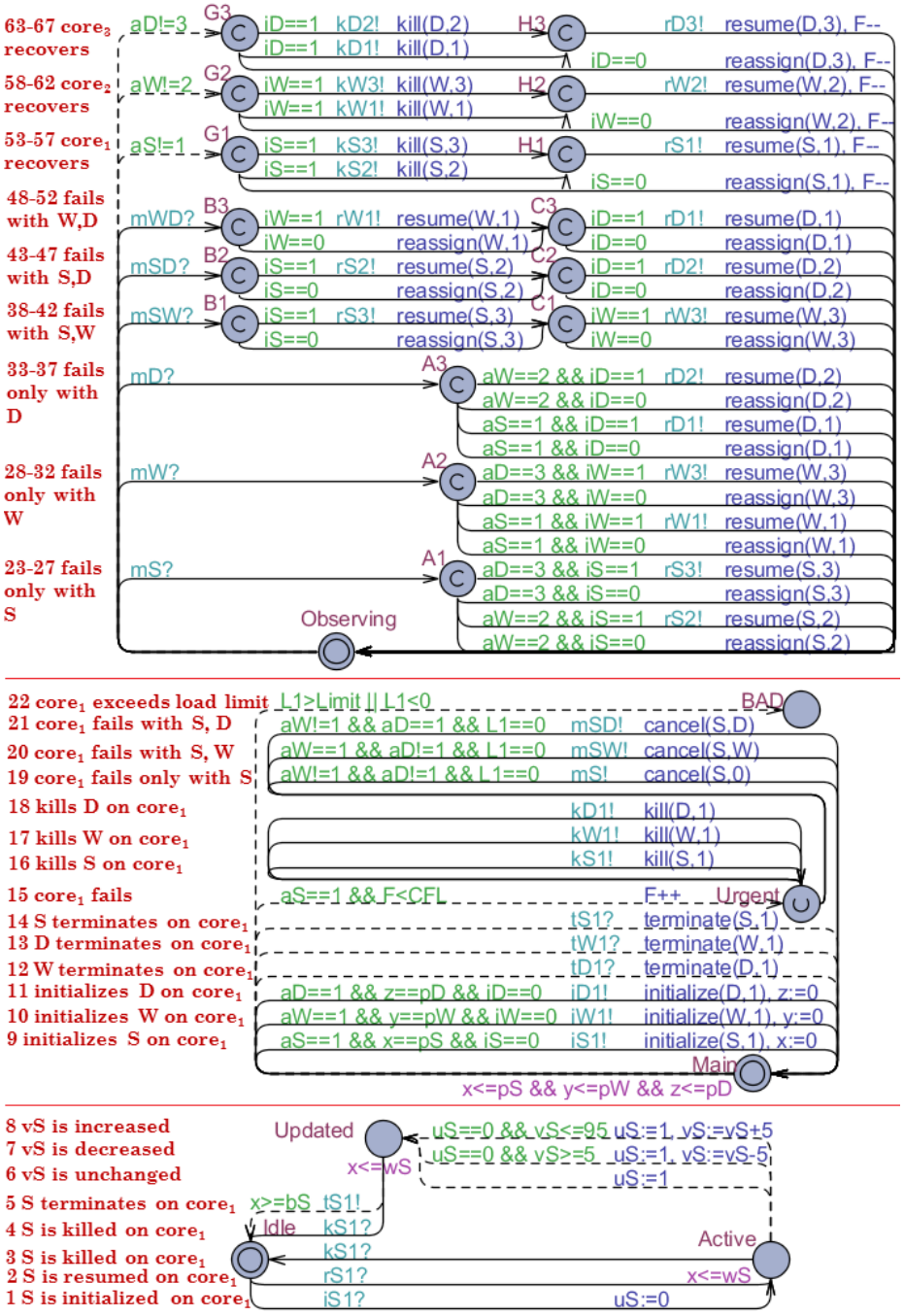

Fig. 3. Automata $\mathsf{core}_1.\mathtt{S}$ (in the bottom), core_1 (in the middle), `service` (in the top) of the concrete model (comments are on the left)

to its operational primary core (core_1). At runtime, the reconfiguration services use a distributed monitoring system to identify these lists because no failed core can broadcast a message. An unsafe location `BAD` becomes reachable when the current worst-case load on core_1 exceeds the load limit of core_1 because of the failure of some other core(s) (edge 22). This prevents the synthesis algorithm from producing a control strategy that would require exceeding the load limits.

4.3 Service Automaton

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

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

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

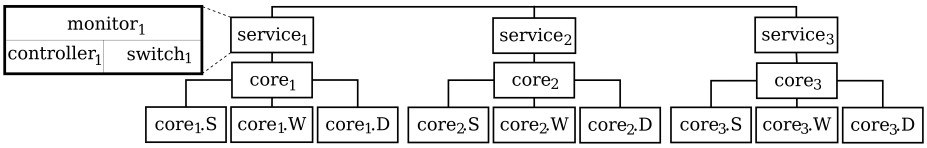

Fig. 4. Architecture of system_1 at runtime

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

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

5 Synthesis

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 switch to cancel and reinstate the periodic execution of non-critical tasks. All the distributed controllers of a system differ from each other—but complement each other in a way that they all together work similarly with a central controller, which is synthesized by analyzing the timed game model of Sect. 4. Figure 4 presents our proposed architecture of system_1.

5.1 Controller Synthesis

We perform a controller synthesis for the monolithic model of Sect. 4 against a safety objective "A[] not (core_1.BAD or core_2.BAD or core_3.Bad)", meaning that there is a strategy to always avoid locations core_1.BAD, core_2.BAD, and core_3.BAD. If the property holds, the strategy—which is our central controller—is automatically synthesized by a timed game solver.

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 Fig. 1 the controller has non-deterministic choices at system-state s_4 when only $core_2$ fails and the other two cores are operational (edges 28–32). A strategy removes non-determinism for the controller. By directing the controller to take the correct paths, a strategy plays a crucial role when in the model some paths guarantee satisfaction of a property (say reallocating task `W` to $core_3$ at system-state s_5 in Fig. 1) and some paths do not (say reallocating `W` to $core_1$). For example, when $core_2$ fails (edges 28–32) a strategy (or the central controller) may say, "`if the system-state fulfills condition` X `then reallocate task W to` $core_3$`, otherwise to` $core_1$"; then the distributed controller of this portion (edges 28–32) for $core_3$ is "`if the system-state fulfills condition` X `then reallocate task W to` $core_3$"; and the distributed controller of this portion (edges 28–32) for $core_1$ is "`if the system-state does not fulfill condition` X `then reallocate task W to` $core_1$". Thus, deriving the distributed controllers from the central controller is a mechanical process and cannot fail.

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, then 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 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. Hence, MCFL for $system_1$ for configuration C1 is 1.

5.2 Service Synthesis

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

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.

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.

5.3 Scalability

The scalability of our service synthesis process mostly depends on the 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. Moreover, for many configurations (C3–C6 in Table 1) the solver runs out of memory during analysis. 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.

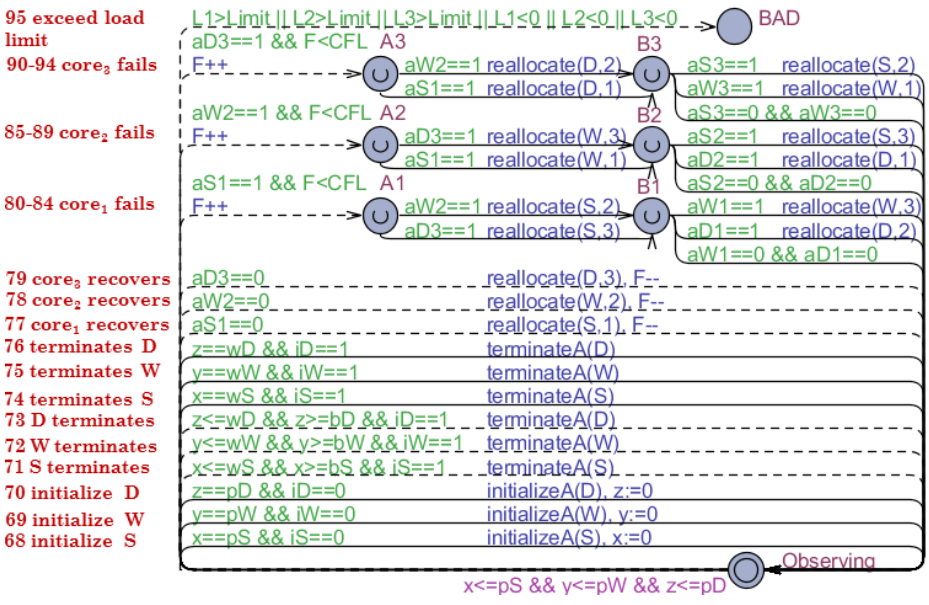

Fig. 5. The abstract model (comments are on the left)

	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			C F L	Comparison			
																				concrete model		abstract model	
	S	W	D	S	W	D	S	W	D	S	W	D	S	W	D	S	W	D		time	size	time	size
C1	10	10	10	5	5	5	4	4	4	60	45	5	10	80	5	10	5	85	2	No controller exists			
																			1	94.20	290663	0.08	73
C2	10	10	10	5	5	5	0	0	0	60	45	5	10	80	5	10	5	85	2	No controller exists			
																			1	115.71	296524	0.11	107
C3	10	15	20	5	5	5	0	0	0	60	45	5	10	80	5	10	5	85	2	No controller exists			
																			1	Out of memory		0.14	242
C4	10	15	20	5	5	5	0	0	0	60	35	5	10	80	5	10	5	85	2	Out of memory		0.25	712
																			1			0.14	266
C5	10	15	20	5	5	5	0	0	0	43	37	7	11	67	19	23	13	59	2	Out of memory		0.25	712
																			1			0.14	266
C6	10	15	20	5	5	5	0	0	0	43	37	59	11	67	39	23	13	59	2	No controller exists			
																			1				

Table 1. Comparisons of the two models with respect to their controller synthesis time (in sec.) and the strategy size (in KB), for different configurations (release period, WCET, and BCET have abstract time units; and loads are in % of the respective core)

The main purpose of the strategy is to resolve non-determinism among enabled controllable transitions in a way that guarantees satisfaction of the desired property. Hence, one can abstract away every detail from a timed game model that does not contribute to the non-determinism (or to the property). For instance, task specific activities and their non-deterministic updates of the tasks, which do not have any impact on our property, can be removed from a timed game model of system_1. Using such aggressive abstractions we construct the abstract model of system_1. Presented in Fig. 5, the model has only one automaton.

The abstract model uses all the modeling abstractions and system parameters of Sect. 4. Explicitly it models only task initializations (edges 68–70), task terminations (edges 71–76), core failures (edges 77–79), core recovery (edges 80–94), and safety violations (edge 95). Like task killings and resumptions, task initializations and terminations change the load on a core; thus they play an important role in the required property (or the safety checking). The invariant is used to release or initialize the tasks periodically. While a task termination within the WCET is forced by allowing an additional controllable transition (edges 74–76). Reallocation is a function which combines task killings, reassignments, and resumptions (edges 77–94). The model uses nine Boolean variables `aS1`, `aW1`, `aD1`, `aS2`, `aW2`, `aD2`, `aS3`, `aW3`, and `aD3` to keep track the currently assigned tasks to cores: the value of `aS1` (resp., `aW1`, `aD1`) is 1 when the periodic execution of task `S` (resp., `W`, `D`) is assigned to core core_1, otherwise the value is 0; similarly, `aS2` (resp., `aW2`, `aD2`) is 1 if and only if the periodic execution of task `S` (resp., `W`, `D`) is assigned to core core_2. If both the concrete model and the abstract model use a variable or constant then it is used for the same purpose; for example, variable `iS` in both the models is used to identify when task `S` is initialized.

Experimental Results We analyze the two models with different configurations. All the analyses and controller syntheses were performed by Uppaal Tiga-0.17 on a PC with an Intel Core i3 CPU at 2.4 GHz, 4 GB of RAM, and running

64-bit Windows 7. Table 1 shows that the abstract model improves the scalability dramatically. Other than aggressive abstraction, it encodes the whole model into only an automaton to avoid parallel composition, because parallel composition typically increases the size of the state space very rapidly. The larger the difference between WCET and BECT the longer the analysis time, and the sparser the strategy (configuration C1 vs. configuration C2). The smaller the least common denominator of the clock ranges (or the execution times and release periods) the smaller state space, the shorter analysis time, and the more compact strategy (C2 vs. C3). However, variations in the least common denominator of other variables, such as different loads, do not have any significant impact on the analysis (C4 vs. C5). Uppaal Tiga takes less time and generates a smaller strategy when the environment has less "freedom", as in, e.g. the case of a smaller value for CFL (C4, C5). The MCFL of system system_1 depends on its configuration: the MCFL is 1 for the first three configurations (C1–C3); 2 for the next two configurations (C4, C5); and 0 for the last configuration (C6).

5.4 Discussion

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

We have assumed that an initialized task reallocated from a failed core should resume in the same state. If this is not required, i.e., a task can start from initial state on the new core at its next release period, then the model can be simplified, by removing the edges modeling resumption. In the future, we will show how to synthesize reconfiguration services that also ensure the maximal utilization of the processing capacity at fault-time. We have not investigated the synthesis process for a scheduler with a dynamic allocation yet.

6 Conclusion

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. To the best of our knowledge, this is the first case study applying timed games to the synthesis reconfiguration services for fault-tolerance.

Acknowledgements We would like to express our gratitude to GM engineers and scientist, especially Joseph D'Ambrosio for introducing the problem, Thomas E Fuhrman and Ramesh S for a series of meetings on the fault models of automotive mixed-criticality multi-core systems, Soheil Samii for discussions on the possibilities of different reconfiguration models, and many others who contributed in the initial phase of this project. We also thank Alexandre David for his help with Uppaal Tiga.

References

1. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems. In: STACS. (1995)
2. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: SSC. (1998)
3. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: The element of surprise in timed games. In: CONCUR. (2003)
4. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Didier, L.: UPPAAL-Tiga: time for playing games! CAV (2007)
5. Ehlers, R., Mattmüller, R., Peter, H.J.: Synthia: Verification and synthesis for timed automata. In: CAV. (2011)
6. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994)
7. Waez, M.T.B., Dingel, J., Rudie, K.: A survey of timed automata for the development of real-time systems. CSR (2013)
8. Tripakis, S.: Fault diagnosis for timed automata. In: Formal Techniques in Real-Time and Fault-Tolerant Systems. (2002)
9. Bouyer, P., Chevalier, F., D'Souza, D.: Fault diagnosis using timed automata. In: FoSSaCS. (2005)
10. Waszniowski, L., Krákora, J., Hanzálek, Z.: Case study on distributed and fault tolerant system modeling based on timed automata. J. Syst. Softw. (2009)
11. Lv, M., Yi, W., Guan, N., Yu, G.: Combining abstract interpretation with model checking for timing analysis of multicore software. In: RTSS. (2010)
12. Dalsgaard, A.E., Laarman, A., Larsen, K.G., Olesen, M.C., van de Pol, J.: Multi-core reachability for timed automata. FORMATS (2012)
13. Socci, D., Poplavko, P., Bensalem, S., Bozga, M.: Modeling mixed-critical systems in real-time BIP. ReTiMiCS (2013)
14. David, A., Grunnet, J., Jessen, J., Larsen, K., Rasmussen, J.: Application of model-checking technology to controller synthesis. In: Formal Methods for Components and Objects. (2012)
15. Kaynar, D.K., Lynch, N.A., Segala, R., Vaandrager, F.W.: The Theory of Timed I/O Automata. SLCS. Morgan & Claypool Publishers (2006)
16. David, A., Larsen, K.G., Legay, A., Nyman, U., Wąsowski, A.: Timed I/O automata: a complete specification theory for real-time systems. HSCC (2010)

A Appendix

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

Table 2. Variables in the concrete model

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

Table 3. Constants in the concrete model

Name	From	To	Purpose
iS1	$core_1$	$core_1$.S	To initialize S on $core_1$
iS2	$core_2$	$core_2$.S	To initialize S on $core_2$
iS3	$core_3$	$core_3$.S	To initialize S on $core_3$
iW1	$core_1$	$core_1$.W	To initialize W on $core_1$
iW2	$core_2$	$core_2$.W	To initialize W on $core_2$
iW3	$core_3$	$core_3$.W	To initialize W on $core_3$
iD1	$core_1$	$core_1$.D	To initialize D on $core_1$
iD2	$core_2$	$core_2$.D	To initialize D on $core_2$
iD3	$core_3$	$core_3$.D	To initialize D on $core_3$
rS1	service	$core_1$.S	To resume S on $core_1$
rS2	service	$core_2$.S	To resume S on $core_2$
rS3	service	$core_3$.S	To resume S on $core_3$
rW1	service	$core_1$.W	To resume W on $core_1$
rW2	service	$core_2$.W	To resume W on $core_2$
rW3	service	$core_3$.W	To resume W on $core_3$
rD1	service	$core_1$.D	To resume D on $core_1$
rD2	service	$core_2$.D	To resume D on $core_2$
rD3	service	$core_3$.D	To resume D on $core_3$
kS1	$core_1$, service	$core_1$.S	To kill S on $core_1$
kS2	$core_2$, service	$core_2$.S	To kill S on $core_2$
kS3	$core_3$, service	$core_3$.S	To kill S on $core_3$
kW1	$core_1$, service	$core_1$.W	To kill W on $core_1$
kW2	$core_2$, service	$core_2$.W	To kill W on $core_2$
kW3	$core_3$, service	$core_3$.W	To kill W on $core_3$
kD1	$core_1$, service	$core_1$.D	To kill D on $core_1$
kD2	$core_2$, service	$core_2$.D	To kill D on $core_2$
kD3	$core_3$, service	$core_3$.D	To kill D on $core_3$
iS1	$core_1$.S	$core_1$	To terminate S on $core_1$
iS2	$core_2$.S	$core_2$	To terminate S on $core_2$
iS3	$core_3$.S	$core_3$	To terminate S on $core_3$
iW1	$core_1$.W	$core_1$	To terminate W on $core_1$
iW2	$core_2$.W	$core_2$	To terminate W on $core_2$
iW3	$core_3$.W	$core_3$	To terminate W on $core_3$
iD1	$core_1$.D	$core_1$	To terminate D on $core_1$
iD2	$core_2$.D	$core_2$	To terminate D on $core_2$
iD3	$core_3$.D	$core_3$	To terminate D on $core_3$
mSW	$core_1$, $core_2$	service	To inform that it is assigned to execute S and W
mSD	$core_1$, $core_3$	service	To inform that it is assigned to execute S and D
mWD	$core_2$, $core_3$	service	To inform that it is assigned to execute W and D
mS	$core_1$	service	To inform that it is assigned to execute S
mW	$core_2$	service	To inform that it is assigned to execute W
mD	$core_3$	service	To inform that it is assigned to execute D

Table 4. Actions in the concrete model

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

Table 5. Variables in the abstract model

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

Table 6. Constants in the abstract model

```
const int S=1,W=2,D=3;
const int Limit=100;
const int pS=10,pW=15,pD=20;
const int bS=0,bW=0,bD=0;
const int wS=5,wW=5,wD=5;
const int CFL=2;
const int lS1=43,lW1=37,lD1=59;
const int lS2=11,lW2=67,lD2=39;
const int lS3=23,lW3=13,lD3=59;

clock x,y,z;
bool uS=0,uW=0,uD=0;
bool iS=0,iW=0,iD=0;
int [0,3] aS=1,aW=2,aD=3;
int [0,200] L1=0,L2=0,L3=0;
int [0,150] vS=0;
int [0,2] sD=0;
int [0,3] F=0;

chan mSD,mSW,mWD,mS,mW,mD;
chan iS1,iW1,iD1,tS1,tW1,tD1,kS1,kW1,kD1,rS1,rW1,rD1;
chan iS2,iW2,iD2,tS2,tW2,tD2,kS2,kW2,kD2,rS2,rW2,rD2;
chan iS3,iW3,iD3,tS3,tW3,tD3,kS3,kW3,kD3,rS3,rW3,rD3;
```

Fig. 6. Declarations in the concrete model for configuration C6 when CFL is 2

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

Fig. 7. Functions `initialize`, `terminate`, `kill`, and `cancel` in the concrete model

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

Fig. 8. Functions `resume` and `reassign` in the concrete model

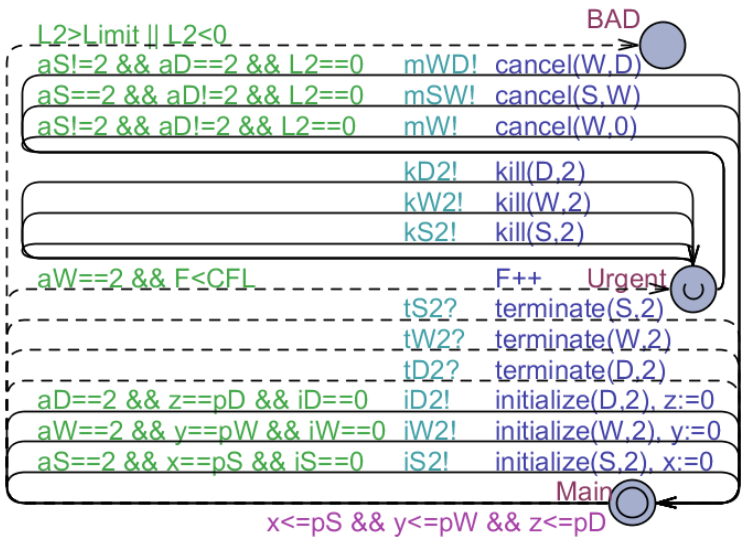

Fig. 9. Automaton core_2 in the concrete model

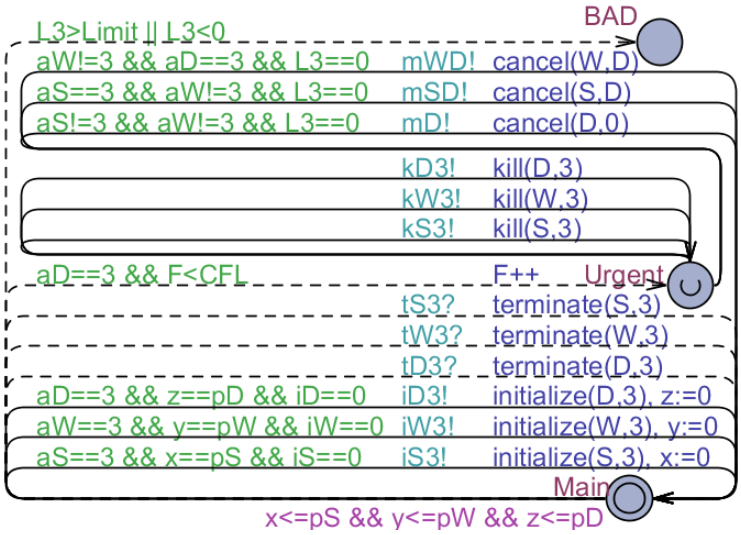

Fig. 10. Automaton core_3 in the concrete model

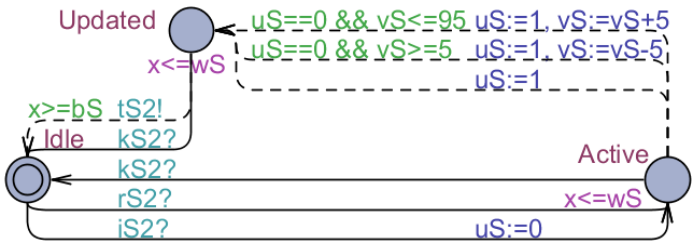

Fig. 11. Automaton core_2.S in the concrete model

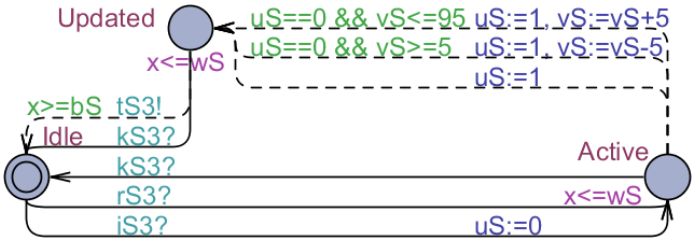

Fig. 12. Automaton core_3.S in the concrete model

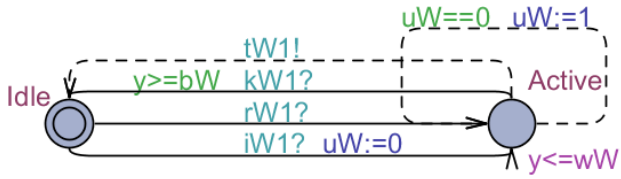

Fig. 13. Automaton core_1.W in the concrete model

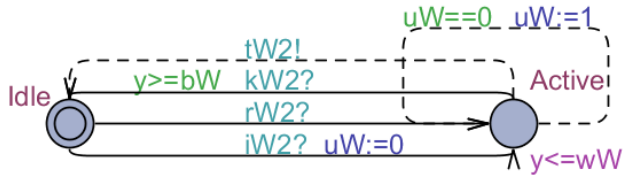

Fig. 14. Automaton core_2.W in the concrete model

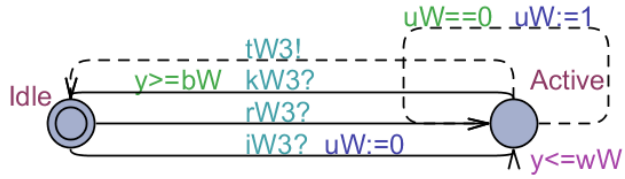

Fig. 15. Automaton core_3.W in the concrete model

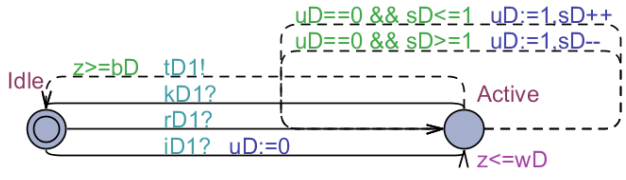

Fig. 16. Automaton core_1.D in the concrete model

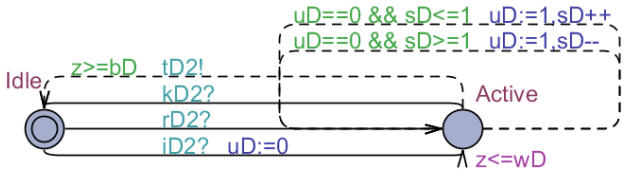

Fig. 17. Automaton core_2.D in the concrete model

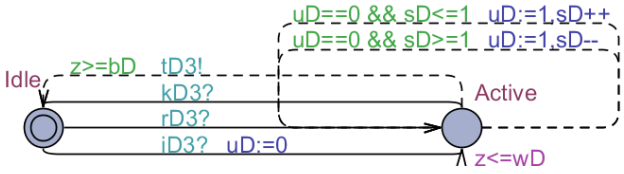

Fig. 18. Automaton core_3.D in the concrete model

```
const int CFL=2;
const int Limit=100;
const int S=1,W=2,D=3;
const int pS=10,pW=15,pD=20;
const int wS=5,bS=0;
const int wW=5,bW=0;
const int wD=5,bD=0;
const int lS1=43,lW1=37,lD1=59;
const int lS2=11,lW2=67,lD2=39;
const int lS3=23,lW3=13,lD3=59;

clock x,y,z;
int F=0;
int L1=0,L2=0,L3=0;
bool aS1=1,aS2=0,aS3=0;
bool aW1=0,aW2=1,aW3=0;
bool aD1=0,aD2=0,aD3=1;
bool iS=0,iW=0,iD=0;
```

Fig. 19. Declarations in the abstract model for configuration C6 when CFL is 2

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

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

Fig. 20. Functions `initializeA` and `terminateA` in the abstract model

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

Fig. 21. Function `reallocate` in the abstract model