Contract-based Compositional Scheduling Analysis for Evolving Systems *

Tayfun Gezgin[1], Stefan Henkler[1], Achim Rettberg[2], and Ingo Stierand[2]

[1] Institute for Information Technology (OFFIS)
[2] Carl von Ossietzky University Oldenburg

Abstract. The objective of this work is the analysis and verification of distributed real-time systems. Such systems have to work in a timely manner in order to deliver the desired services. We consider a system architecture with multiple computation resources. The aim is to work out a compositional state-based analysis technique to determine exact response times and to validate end-to-end deadlines. Further, we consider such systems in a larger context, where a set of systems work in a collaborative and distributed fashion. A major aspect of such collaborative systems is the dynamic evolution. New systems can participate, existing systems may leave because of failures, or properties may change. We use contracts to encapsulate systems which work in a collaborative manner. These contracts define sound timing bounds on services offered to the environment. When some systems evolve, only those parts which changed need to be re-validated.

Keywords: Compositional Analysis, Real-Time Systems, Scheduling Analysis, Model Checking, Abstraction Techniques

1 Introduction

For safety-critical distributed systems it is crucial that they adhere to their specifications, as the violation of a requirement could lead to very high costs or even threats to human life. One crucial aspect for safety critical systems is that they have to work in a timely manner. Therefore, in order to develop safe and reliable systems, rigorous analysis techniques of timing-dependent behaviour are necessary. In literature, basically there are two approaches for scheduling analysis of distributed real-time systems. The classical approach is a holistic one, as it was worked out by e.g. Tindell and Clark [12]. Here, local analysis is performed

* This work was partly supported by European Commission funding the Large-scale integrating project (IP) proposal under ICT Call 7 (FP7-ICT-2011-7) Designing for Adaptability and evolutioN in System of systems Engineering (DANSE) (No. 287716), and by the German Research Council (DFG) as part of the Transregional Collaborative Research Center 'Automatic Verification and Analysis of Complex Systems' (SFB/TR 14 AVACS).

evaluating fixed-point equations. The analysis is very fast and is able to handle large systems evaluating performance characteristics like time and memory consumption. Unfortunately, it delivers very pessimistic results when inter-ECU task dependencies exist. In [10] activation pattern for tasks are described by upper and lower arrival curves realizing a more *compositional* analysis method. Based on this work a compositional scheduling analysis tool, called SymTA/S, was created by SymtaVision. The concept has been developed by Kai Richter et.al. [8], and was improved and extended in several works, e.g. [9]. The main idea behind SymTa/S is to transform event streams whenever needed and to exploit classical scheduling algorithms for local analysis. This concept is very fast and is able to handle large systems, but typically yields pessimistic results.

The second approach is based on model checking, and has been illustrated for example in [5, 4]. Here, all entities like tasks, processors, and schedulers are modeled in terms of timed automata. The advantage of this approach is that one gets exact solutions with respect to the modeled scheduling problem. As the state space of the analyzed system is preserved, checking complex characteristics like safety properties is possible. Unfortunately, the approach is not scalable, as the state space of the whole architecture is generated in one single step.

Our approach for scheduling analysis combines both analytical and model checking methods. Analogous to [5] we consider the full state space for analysis, where all inter-leavings and task dependencies are preserved. For this, the state space of the entire system architecture is constructed in a compositional manner. Based on the state space of a resource, response times are determined. Further, we propose a concept in order to handle timing properties of evolving systems. Evolving systems are such systems, where parts can change during run-time. Changes can occur due to failures and reconfigurations, or when new tasks are allocated to existing resources. When such a part of the system changes, it is desirable to only re-validate this part locally without the rest of the system.

Next, we present in Section 2 the foundation of our approach. In Section 3 we introduce operations on symbolic transition systems in order to realize our compositional analysis. In Section 3.3, we present our state-based analysis technique. The contract based approach for evolving systems is illustrated in Section 4, and finally we give a conclusion.

2 Fundamentals

In this work we consider system architectures consisting of sets of processing units (ECU). A set of tasks is allocated to each ECU. Our goal is to deter-

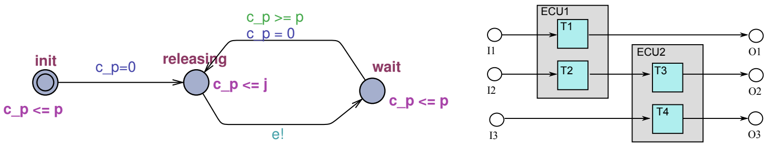

Fig. 1. Left: characterization of event streams; Right: example architecture

mine for each task corresponding response times, and whether all task deadlines and end-to-end deadlines are satisfied. More specific, a task is a tuple $\tau = (bcet, wcet, d, pr)$, where $bcet, wcet \in \mathbb{N}_{\geq 0}$ are the best and worst case execution times with respect to the allocated ECU with $bcet \leq wcet$, $d \in \mathbb{N}_{\geq 0}$ is its deadline determining the maximal allowed time frame from release time to task termination, and $pr \in \mathbb{N}_{\geq 0}$ is the fixed priority of the task. We will refer to the elements of a tasks by indexing, e.g. $bcet_\tau$ for task τ. The set of all tasks is called T. Independent tasks are triggered by events of a corresponding event stream (ES). An event stream $ES = (p, j)$ is characterized by a period p and a jitter j with $p, j \in \mathbb{N}_{\geq 0}$. Such streams can be characterized by upper and lower occurrence curves as introduced in the real-time calculus [11] or timed automata (introduced in the next section) like illustrated in the left part of Figure 1. In this work we will restrict to event streams where $j_\tau < p_\tau$ for all $\tau \in \mathsf{T}$. Also more general event streams with bursts would be possible. Automata for such event streams were presented in [6]. Task dependencies are captured by connecting corresponding tasks, like e.g. in Figure 1 where t_3 depends on t_2.

Each ECU in a system is modeled by the tuple $ecu = (\mathcal{T}, Sch, \mathcal{R}, \mathcal{S}, \mathcal{A})$. A mapping $\mathcal{T} : \mathsf{T} \to \mathbb{B}$ determines the set of tasks that are allocated to the ECU. For each ECU, a scheduling policy Sch is given. Three additional state dependent functions provide dynamic book keeping needed to perform scheduling analysis: (i) a ready map $\mathcal{R} : \mathsf{T} \to \mathbb{B}$ determines tasks, which are released but to which no computation time has been allocated up to now, (ii) a start delay map $\mathcal{S} : \mathsf{T} \to [t_1, t_2]$ with $t_1, t_2 \in \mathbb{N}_{>0}$ which determines the delay interval for a task getting from status *released* to *run*, and (iii) an active task map $\mathcal{A} : \mathsf{T} \to [t_1, t_2]$ with $t_1, t_2 \in \mathbb{N}_{>0}$ which determines the interruption times of tasks. This map is ordered and the first element determines the currently running task.

2.1 Timed Automata: Syntax and Semantics

Timed automata [1] are finite automata extended with a finite set of real-valued variables called clocks. Here, we define syntax and semantics of timed automata as employed by Uppaal. Uppaal adapts timed safety automata introduced in [7]. In such automata, progress is enforced by means of local invariants. States (or locations) may be associated with a timing constraint defining upper bounds on clocks. Let C be a set of clocks. A clock constraint is defined by the syntax $\varphi ::= \ c_1 \sim t \mid c_1 - c_2 \sim t \mid \varphi \wedge \varphi$, where $c_1, c_2 \in C$, $t \in \mathbb{Q}_{\geq 0}$ and $\sim \in \{\leq, <, =, >, \geq\}$. The set of all clock constraints over the set of clocks C is denoted by $\Phi(C)$. Valuation of a set of clocks C is a function $\nu : C \to \mathbb{R}_+$ assigning each clock in C a non-negative real number. We denote $\nu \models \varphi$ the fact that a clock constraint φ evaluates to true under the clock valuation ν. We use 0_C to denote the clock valuation $\{c \mapsto 0 \mid c \in C\}$, abbreviate the time shift by $\nu + d := \nu(c) + d$ for all $c \in C$, and define clock resets for a set of clocks $\varrho \subseteq C$ by $\nu[\varrho \mapsto 0]$ with $\nu[\varrho \mapsto 0](c) = 0$ if $c \in \varrho$, and $\nu[\varrho \mapsto 0] = \nu(c)$ else.

Definition 1 (Timed Automaton).
A Timed Automaton *(TA) is a tuple $A = (L, l^0, \Sigma, C, R, I)$ where*

- *L is a finite, non-empty set of locations, and $l^0 \in L$ is the initial location,*
- *Σ is a finite alphabet of channels, and C is a finite set of clocks,*

- *$R \subseteq L \times \Sigma \times \Phi(C) \times 2^C \times L$ is a set of transitions. A tuple $r = (l, \sigma, \varphi, \varrho, l')$ represents a transition from location l to location l' annotated with the action σ, constraint φ, and a set ϱ of clocks which are reset.*
- *$I : L \rightarrow \Phi(C)$ is a mapping which assigns an invariant to each location,*

The semantics of timed automata is given by timed transition systems.

Definition 2 (Timed Transition System). *Let $A_i = (L_i, l_i^0, \Sigma_i, C_i, R_i, I_i)$ with $i \in \{1, ..., n\}$ be a network of timed automata with pairwise disjoint sets of clocks and alphabets. The semantics of such a network is defined in terms of a timed transition system $\mathcal{T}(A_1 \parallel ... \parallel A_n) = (Conf, Conf^0, C, \Sigma, \rightarrow)$, where*

- *$Conf = \{(l, \nu) \mid l \in L_1 \times ... \times L_n \ \wedge \ \nu \models \bigwedge_{j=1}^{n} I_j(l_j)\}$ is the set of configurations, and $Conf^0 = (l^0, 0_C)$, where $l^0 = (l_1^0, ..., l_n^0)$ is the initial location and 0_C is the initial clock valuation,*
- *$C = C_1 \cup ... \cup C_n$, $\Sigma = \Sigma_1 \cup ... \cup \Sigma_n$,*
- *$\rightarrow \subseteq Conf \times (\Sigma \cup \mathbb{R}_+) \times Conf$ is the transition relation. A transition $((l, \nu), \lambda, (l', \nu'))$, also denoted by $(l, \nu) \xrightarrow{\lambda} (l', \nu')$, has one of the following types.*
  - *A flow transition $(l, \nu) \xrightarrow{t} (l, \nu + t)$ with $t \in \mathbb{R}_+$ can occur, if $\nu + t \models \bigwedge_{j=1}^{n} I_j(l_j)$.*
  - *A discrete transition $(l, \nu) \xrightarrow{\lambda} (l', \nu')$ with $l' = l[l_i \rightarrow l_i']$ and $\lambda \in \Sigma$ can occur, if for some $i \in \{1, ..., n\}$ it holds that $(l_i, \lambda_i, \varphi_i, \varrho_i, l_i') \in R_i$, such that $\nu \models \varphi_i$, $\nu' = \nu[\varrho_i \mapsto 0]$ and $\nu' \models \bigwedge_{j=1}^{n} I_j(l_j')$.*

The function $l[l_i \rightarrow l_i']$ for a location vector $l = (l_1, ..., l_i, ..., l_n)$ represents the location vector $l = (l_1, ..., l_i', ...l_n)$. As the set of configurations is infinite, [1] gives a finite representation which is called region graph. In [2] a more efficient data structure called zone graph was presented. A zone represents the maximal set of clock valuations satisfying a corresponding clock constraint. Let $g \in \Phi(C)$ be a clock constraint, the induced set of clock valuations $D_g = \{\nu \mid \nu \models g\}$ is called a clock zone. Let $D^{\uparrow} = \{\nu + d \mid \nu \in D \wedge d \in \mathbb{R}_+\}$ and $D[\varrho \rightarrow 0] = \{\nu[\varrho \mapsto 0] \mid \nu \in D\}$. The finite representation of a timed automaton is given by a symbolic transition system.

Definition 3 (Symbolic Transition System). *Let A be a network of timed automata with pairwise disjoint sets of clocks and alphabets. The symbolic transition system (zone graph) of A is a tuple $STS(A) = (S, S^0, \rightarrow)$ where*

- *$S = \{\langle l, D_{\varphi} \rangle \mid l \in L_1 \times ... \times L_n, \varphi \in \Phi(C)\}$ is the symbolic state set, and $S^0 = \langle l^0, 0_C \rangle$ is the initial state,*
- *$\rightarrow \subseteq S \times S$ is the symbolic transition relation with*
  - *$\langle l, D \rangle \rightarrow \langle l, D^{\uparrow} \cap D_{I(l)} \rangle$, where $I(l) = \bigwedge_{j=1}^{n} I_j(l_j)$*
  - *$\langle l, D \rangle \rightarrow \langle l', (D \cap D_{\varphi_i})[\varrho_i \rightarrow 0] \cap D_{I(l')} \rangle$ where $l' = l[l_i \rightarrow l_i']$, if there is a $i \in \{1, ..., n\}$ such that $(l_i, \lambda_i, \varphi_i, \varrho_i, l_i') \in R_i$.*

Note that for the general case some so called normalization operations on zones are necessary. If we build the symbolic transition system for an automaton containing clocks without a ceiling, i.e. some maximal reachable upper bound, it will lead to infinite sets of symbolic states. Nevertheless, for our cases the above definition will be sufficient as we always will have ceilings for all clocks. Please refer to [2] for more details on zone normalization operations.

3 Compositional Analysis

We build the state spaces - i.e. the symbolic transition systems (STS) - of each resource successively. These state spaces contain the response times of the allocated tasks. To construct the STS of a resource, besides the behaviour of the *scheduler* and characteristics of the allocated tasks, an *input STS* describing the activation times of the tasks is necessary. Generally, the inputs of the tasks are originated from different sources, such that multiple input STSs are given and we have first to build the product of these STS. As an example, consider Figure 1: to compute the STS of $ECU2$, we need the activation behaviour of task t_4, which is given by the event stream $I3$, while the input for t_3 is given by the output STS of resource $ECU1$. Further, the input STS of a resource can include behaviour which is not relevant for the computation of the resource STS. In the above example, to compute the STS of *ECU2*, only the part of the state space of *ECU1* containing the behaviour of task t_2 is relevant. We can skip the part of the state space, in which detailed information about task t_1 is present.

In the following, we will first introduce both operations on STS, i.e. the abstraction operation in Section 3.1 and the product computation in Section 3.2. The construction of the STS of a resource is detailed in Section 3.3.

3.1 State Abstraction

In general, an abstraction function is defined as $\alpha : S \rightarrow S'$, where $S' \subseteq S$ for a state set S. In the context of scheduling analysis we will define in the following the state space of our considered problem domain. Then, we will introduce two specific abstraction functions, one operating on zones and one on locations. These two abstraction functions will then be combined and applied to our problem domain.

Considered State Space To capture the initial non-determinism of the input event streams of n independent tasks of a ECU as defined in the previous section, the corresponding STS consists of 2^n locations. Let $L = \{0, (l^1), ...(l^n), (l^1, l^2), ..., (l^1, ..., l^n)\}$ be a set of discrete locations over index set $I = \{1, ..., n\}$. The location (l^i) indicates that an instance of task τ_i has already been released at least once. Analogously, location $(l^1, ..., l^n)$ indicates that all tasks have already been released at least once.

Besides the set of locations the considered state set is defined over clock valuations over a set of clocks C. For each *independent* task τ two types of clocks are needed, i.e. (i) clocks which trace the periodical activation of each task ($c_p(\tau)$ for task τ), and (ii) clocks which trace the time frame from releasing a task up to the finish of computation ($c_{active}(\tau)$). In order to capture overlapping task activations, i.e., where multiple *task instances* t_i of task τ may be active at the same time, multiple clocks $c_{active}(t_i)$ exist, one for each task instance. We need multiple clocks as we rely on using simple *clocks* in order to realize our scheduling analysis with preemption, i.e. we cannot change the derivative of a clock. Otherwise, we would have so called stopwatch automata where a stopwatch is used to track the allocated execution times of tasks. For this class of automata the reachability problem is known to be undecidable [3]. As we need to use one separate clock per task instance, we need to know a priory the maximal number of possible parallel activations of one task. For *dependent* tasks we need only the second class of clocks as we do not have to trace the activation times. For a task set T we will denote with $clk(\mathsf{T})$ the set of clocks of all tasks in T. As an example, consider the right part of Figure 1, where we have the periodical clocks $c_p(t_1), c_p(t_2), c_p(t_3)$, and allocated computation time clocks $c_{active}(t_1), c_{active}(t_2), c_{active}(t_3), c_{active}(t_4)$.

Abstraction on Zones Let $C' \subseteq C$. For a constraint $g \in \Phi(C)$ let $g_{|C'}$ be the constraint, where all propositions containing clocks of the set $C \backslash C'$ are removed. Analogously, for a constraint $g \in \Phi(C')$ let $g_{|C}$ be the constraint extended with propositions containing clocks in $C \backslash C'$. The extension is defined in such a way, that it does not affect the original zone, i.e. $D_g = (D_{g_{|C}})_{|C'}$ and the new constraints are of the form $0 \leq c \leq \infty$ forall $c \in C \backslash C'$. For example, consider the sets $C = \{c_1, c_2\}, C' = \{c_2\}$ and the constraint $g = c_1 - c_2 \leq 3 \wedge c_1 \leq 5 \wedge c_2 \leq 1$. Then $g_{|C'} = c_2 \leq 1$. Further, we have $(g_{|C'})_{|C} = c_2 \leq 1 \wedge 0 \leq c_1 \leq \infty$. For a zone $D = \{\nu \mid \nu \models g\}$ defined over C we define the zone projection operation $D_{|C'} = \{\nu \mid \nu \models g_{|C'}\}$ accordingly. Note that we have $D_g \subseteq D_{g|C'}$.

Abstraction on Locations Let a set of locations L over index set I be given. For $I' \subseteq I$ let $\alpha_{I'}(L)$ be the set of locations over index set I', where locations with indexes not in I' are left out. For this, consider for example $I' = I \backslash \{i\}$. Then $\alpha_{I'}(l^1, ..., l^i, ..., l^n) = (l^0, ..., l^{i-1}, l^{i+1}, ..., l^n)$. As abbreviation we will directly use tasks instead of explicit indexes, e.g. $\alpha_{\{\tau_i, \tau_j\}} := \alpha_{i,j}$.

Abstraction on States of Symbolic Transition Systems With the introduced abstraction functions on both clock zones and sets of locations we can now define the abstraction function which abstracts sets of states of a STS. Note that these states are tuple over locations and zones. Let $T \subseteq \mathsf{T}_e$, where T_e is the set of tasks allocated to ECU e. The abstraction function abstracts from the state set of the STS of ECU e to the parts where only information about a chosen sub-task set T is kept:

$$\alpha_T : \langle l, D \rangle \rightarrow \langle \alpha_T(l), D_{clk(T)} \rangle. \tag{1}$$

This abstraction function induces the following over-approximated STS:

Definition 4. *Let STS $A = (S, S_0, \rightarrow)$ be a STS over a task set T. The induced abstraction for $T \subseteq \mathsf{T}$ from α_T is the STS $A' = (S', S'_0, \rightarrow')$ with*

- *$S' = \alpha_T(S)$ is the induced set of abstract states, and $S'_0 = \alpha_T(S_0)$ is the initial abstract state,*
- *$\rightarrow' \subseteq S' \times S'$ the abstract transition relation, where $a \rightarrow' b$ iff there exists a $s_1 \in \alpha_T^{-1}(a)$ and $s_2 \in \alpha_T^{-1}(b)$ such that $s_1 \rightarrow s'_2$.*

Due to the definition of the transition relation it is obvious that this abstraction yields an over-approximation of the original STS.

3.2 Product Construction

Let $A_i = (S_i, S_i^0, \rightarrow_i)$ for $i = \{1, ..., n\}$ be a set of STSs over disjoint clock sets C_i and alphabets Σ_i. In the following, we define the product construction $A = A_1 \times ... \times A_n$, which is a STS over clock set $C = C_1 \cup ... \cup C_n$. For each created state of the product STS we need to keep track from which input states, i.e. states of the input STSs $A_1, ..., A_n$, it resulted. For this, we introduce for each A_i the function ξ_i that maps each product STS state to a state of A_i. The initial state of the product is given by

$$\langle l^0, 0_C\rangle = \langle (l_1^0, ..., l_n^0), 0_{C_1} \cup ... \cup 0_{C_n}\rangle \tag{2}$$

where $\langle l_i^0, 0_{C_i}\rangle$ is the initial state of A_i. Note that $\xi_i(\langle l^0, 0_C\rangle) = \langle l_i^0, 0_{C_i}\rangle$. The time successor $\langle l, D'\rangle$ of state $\langle l, D\rangle$ is then determined by

$$\langle l, D'\rangle = \langle l, D^{\uparrow} \cap D'_{1|C} \cap ... \cap D'_{n|C}\rangle \tag{3}$$

where $\xi_i(\langle l, D\rangle) = \langle l_i, D'_i\rangle$ for $i \in \{1, ..., n\}$ are the time successors of the input states. Note that the zones from all STSs are extended to the global clock set C.

Starting from the computed time successor, to compute all possible discrete steps in the product transition system, *each* outgoing transition from *each* STS is tried to be *fired*. In fact, this is also done in Definition 3: whenever the guards of a transition are fulfilled, a discrete transition is enabled and can be fired. This is the case, when the intersection of the zone induced by the guard and the current zone is not empty. The discrete successors of a state $\langle l, D\rangle$ of the product STS are given by the following set:

$$dSucc(\langle l, D\rangle) = \{\langle l[l_i \to l'_i], D'\rangle \mid \langle l_i, D_i\rangle \to \langle l'_i, D'_i\rangle \;\wedge\; D' \neq \emptyset\} \tag{4}$$

where $D' = (D \cap \rho^{-1}(D'_i)_{|C})[\rho(D'_i) \to 0] \cap D'_{i|C}$ and $\langle l_i, D_i\rangle = \xi_i(\langle l, D\rangle)$ for all $i \in \{1, ..., n\}$. The function $\rho(D)$ represents the set of clocks, which are reseted in the corresponding zone and $\rho^{-1}(D)$ represents the symbolic state *before* the reset operation of the corresponding transition has been performed. A discrete step is possible, if the resulting zone is not empty.

3.3 Resource Graph computation

In this section we will illustrate the construction of the state space of a resource. Due to the limited page size we will only sketch the idea of our algorithm here. In the following we will use the functions $i.lb()$ and $i.ub()$ to access the lower and upper bound of an interval i.

Listing 1.1. Main code *computeResourceSTS(STS S_{in}, Configuration c_{in}).*

```
set $\Psi(\langle l^0, 0_C\rangle)$ and $\xi.add(\langle l^0, 0_C\rangle, \langle l^0_{in}, 0_{C_{in}}\rangle)$
while( $\Psi$.size > 0 )
  $\langle l, D\rangle$ = $\Psi$.pop(), $\langle l_{in}, D_{in}\rangle = \xi(\langle l, D\rangle)$
  forall($edge_{in} \in outgoingEdges(\langle l_{in}, D_{in}\rangle)$): computeSuccessor($\langle l, D\rangle, edge_{in}$)
  checkDeadlines()
```

The main algorithm for the computation of a resource STS is illustrated in Listing 1.1. The input parameters are an *input STS* S_{in} describing the task activation times, and some configuration data such as the scheduling policy and informations about the allocated tasks. Analogous to the computation of the product, we need to keep track for each of the resource state, from which input state (i.e. state of the input transition system) it resulted. For this, we introduce the function ξ which maps each resource STS state to an input STS state.

The algorithm starts by creating the initial symbolic state $s_0 = \langle l^0, C_0 \rangle$ of the resource STS. This state is added to a set Ψ, which determines the states, for which successors have to be computed. The initial state $\langle l^0_{in}, 0_{C_{in}} \rangle$ of the input transition system is set as the corresponding input state of s_0.

The possible successors of a state $\langle l, D \rangle$ are determined by the successors of the corresponding input state $\xi(\langle l, D \rangle)$. The computation of the successors of a resource state proceeds analogous to the successor computation of the product STS introduced in the previous section. For this, the algorithm iterates through all outgoing transitions of the corresponding input state defining either the set of tasks, which can be released, or the (single) time successor, and builds a set of corresponding successors of the resource state. If $edge_{in}$ defines the time successor $\langle l_{in}, D_{in} \rangle$, we get the time successor of the current resource state $\langle l, D \rangle$ by computing $D^{\uparrow} \cap D_{in|C}$. If a task instance t is running in state $\langle l, D \rangle$, we further intersect the zone with the response time of the running task, i.e. $c_{active}(t) \leq wcet_t + interruptTimes_t.ub()$.

If else $edge_{in}$ defines a discrete successor which constitutes to a release of a new instance of task τ, we need to perform a case distinction: If there is no running task in the current resource state $\langle l, D \rangle$ (i.e. when the active task map $\mathcal{A}_l$ in location l is empty), we build the successor resource state $\langle l', D' \rangle$ by intersecting the current resource D with the zone of the successor of the corresponding input state. The active task map $\mathcal{A}_l$ then gets the entry $(t_\tau, [0, 0])$.

Else when we have a running task, we have to determine, whether this task can finish its computation before a new instance of the task determined by $edge_{in}$ can be released. This is done as follows.

1. If the running task t cannot terminate, i.e. $c_{active}(t).ub() < bcrt_t$, we try to release a new task instance as defined in the input STS state. For this, the intersection of the current resource state and the successor state of the corresponding input state is computed. Two cases can occur here:
   (a) If the intersection results in an empty zone, the discrete step cannot be taken and the next input edge is considered.
   (b) Else, a new instance of task τ can be released. The active task map $\mathcal{A}_l$ gets the new entry $(t_\tau, [0, 0])$. Then, the next running task wrt. the considered scheduling policy is determined. At least the new state is added to the graph and the set Ψ'.
2. If a running task t will terminate ($c_{active}(t).lb() == wcrt_t$), its execution time is accumulated to all interrupted tasks by incrementing their interrupt times in the active task map $\mathcal{A}_l$. Then we determine the next running task according to the scheduling policy. If we move the task t from the ready map to the active map (i.e. a previously released task which did not get computation time so far), we have to store the time frame from release to start of t in order to correctly determine its allocated execution time. This time frame is given by clock $c_{active}(t)$ and is stored in the start delay map $\mathcal{S}$. Further, we have to reset $c_{active}(t)$ as no computation time has been allocated so far. At least, we are *recursing computeSuccessor*($\langle l', D' \rangle, \tau$).

If $c_{active}(t).ub() \geq bcrt_t \wedge c_{active}(t).lb() \leq wcrt_t$ then both 1) and 2) are executed in this order.

Each state which is generated in the *computeSuccessor* method is added to the set Ψ. For all newly generated states Ψ' we determine whether a deadline is violated as follows.

$$\forall \langle l, D \rangle \in \Psi', t \in \mathcal{A}_l \cup \mathcal{R}_l : \; D_t.ub() \leq d_t. \tag{5}$$

If Equation 5 is violated or no state is left in Ψ, the algorithm terminates.

4 Contract based Analysis

Systems in our context may evolve. Such an evolution is illustrated in Figure 2. The system *System1* is first composed of two resources which deliver some service to its environment. After some time a reconfiguration of this system may occur. In Figure 2 the system is changed to a new decomposition structure consisting of three resources. Such reconfigurations would get necessary if for example some resources fail, loads of the tasks get larger, or new tasks are allocated to the system, such that new resources get necessary. If such systems are annotated by constraints determining the quality of the offered services, internal changes would not affect other systems, as these rely on the quality guarantees of the system. In the above example, a contract consisting of an assumption (A) and a guarantee (G) is annotated to the system. An assumption specifies how the context of the component, i.e. the environment from the point of view of the component, should behave. Only if the assumption is fulfilled, the component will behave as guaranteed. If now a change of a system occurs, we only need to check this part rather than all other systems. When a change occurs, we use the algorithm presented in Section 3.3 to re-validate the contract. This concept can be further extended to parts of Systems of Systems (SoS): if several systems cooperate in order reach some goals and to offer some services to their environment, these cooperating systems can again be annotated by such timing contracts. When a whole system is exchanged by another system in this part of the SoS, again only the STSs of the constituent systems of this part have to be build to re-validate the contract. As we do the analysis in a compositional manner, we further can reuse the STSs of these constituent systems, which are not affected by the changing system. This is for example the case, when they only deliver services to the changed systems and do not adhere on their services.

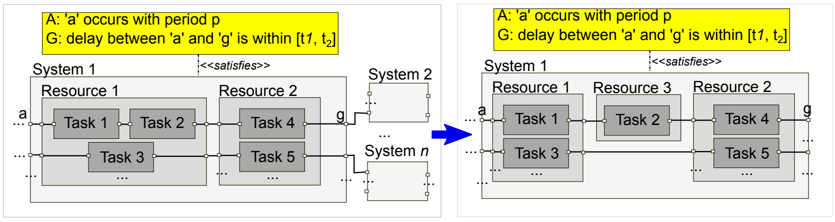

Fig. 2. Changing part of a system.

5 Conclusion and Future Work

In this work we presented a scheduling analysis technique for systems with multiple resources and potential preemptions of tasks. The state space of the entire system architecture is defined by symbolic transitions systems (STS) and is constructed in a compositional manner. For this, we introduced two operations on STSs, namely the product construction and the abstraction on parts of a STS. Further, we proposed a concept in order to handle timing properties of evolving systems. We proposed to encapsulate cooperating system by contracts, such that changes of such a part do not affect other systems. Currently, we are implementing our proposed concept. The implementation so far builds up the STS of the whole architecture and performs the response time analysis in a holistic manner. In future work we will validate our approach and compare it with related tools. We will investigate new abstraction techniques which will further boost the scalability of our approach.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *In Lecture Notes on Concurrency and Petri Nets*, Lecture Notes in Computer Science vol 3098. Springer–Verlag, 2004.
3. Franck Cassez and Kim Larsen. The impressive power of stopwatches. In *In Proc. of CONCUR 2000: Concurrency Theory*, pages 138–152. Springer, 1999.
4. Alexandre David, Jacob Illum, Kim G. Larsen, and Arne Skou. Model-based framework for schedulability analysis using uppaal 4.1. In G. Nicolescu and P.J. Mosterman, editors, *Model-Based Design for Embedded Systems*, pages 93–119, 2009.
5. E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: schedulability and decidability. In *Proceedings of TACAS*. Springer, 2002.
6. M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. In *Parallel and Distributed Processing Symposium*, April 2006.
7. T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1992.
8. Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models.* PhD thesis, Technical University of Braunschweig, Braunschweig, Germany, 2004.
9. J. Rox and R. Ernst. Exploiting inter-event stream correlations between output event streams of non-preemptively scheduled tasks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE, Leuven, Belgium, 2010.
10. L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert. Embedded software in network processors - models and algorithms. pages 416 –434. Springer Verlag London, UK, 2001.
11. L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101 –104 vol.4, 2000.
12. Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40:117–134, April 1994.