Behavioral Modeling and Analysis of Services and Service Compositions

Aida Čaušević Cristina Seceleanu Paul Pettersson
Mälardalen Real-Time Research Centre (MRTC)
Mälardalen University, Västerås, Sweden
{aida.delic,cristina.seceleanu,paul.pettersson}@mdh.se

Abstract

Service-oriented systems have recently emerged as context-independent component-based systems. Unlike components, services can be created, invoked, composed, and destroyed at run-time. Consequently, all services should have a way of advertising their capabilities to the entities that will use them, and service-oriented modeling should cater for various kinds of service composition.

In this paper, we show how services can be formally described by the resource-aware timed behavioral language REMES*, which we extend with service-specific information, such as type, capacity, time-to-serve, etc., as well as boolean constraints on inputs, and output guarantees. Assuming a Hoare-triple model of service correctness, we show how to check it by using the strongest postcondition semantics. To provide means for connecting* REMES *services, we propose a hierarchical language for service composition, which allows for verifying the latter's correctness. The approach is applied on an abstracted version of an intelligent shuttle system, for which we also compute resource-efficient behaviors, and energy-time trade-offs, by model-checking the system's underlying Priced Timed Automata semantic representation.*

I. Introduction

Service-oriented systems (SOS) assume *services* as their basic functional units, with capabilities of being published, invoked, composed and destroyed at runtime. Services are loosely coupled and enjoy a higher level of independence from implementation specific attributes than components do.

An important problem is to ensure the *quality-of-service* (QoS) that can be expected when deciding which service to select out of a number of available services delivering similar functionality. Some of the existing SOS standards support formal analysis [3], [11]–[13] to ensure QoS, but usually it is not straightforward to work out the exact formal analysis model.

Insight into the representation of a service functionality, enabled actions, resource annotations, and possible interactions with other services could be beneficial [7]. For instance, it is good to distinguish between service executions that deliver the result within acceptable time, while using existing resources efficiently, from those who have a slightly better response-time at the expense of using more resources. Hence, in order to fully understand the ways in which services evolve, a *service behavioral description* is required. Such behavior is assumed to be internal to the service, and it is usually hidden from the service user.

To meet the above demands, in this paper, concretely in Section III, we extend the existing resource-aware, timed hierarchical language REMES [15], recalled in Section II, such that it becomes fit for service behavioral modeling. In REMES, a service is modeled by an atomic or composite *mode*, which we enrich with attributes such as service type, capacity, time-to-serve etc., pre- and postconditions, which are exposed at the mode's interface. Exploiting the pre-, postcondition annotations, we show how to check the service correctness by using Dijkstra's and Scholten's strongest postcondition semantics [8].

Since services can be composed at run-time, analyzing the correctness of a service in isolation does not suffice. For example, consider a situation where there exists a request from a user on an online train ticket purchase service. To enable an online train ticket purchase, a set of functionally smaller services must be invoked, composed, and executed (e.g., *display available trains*, *show prices*, *place an order*, *enter customer details*, *check credit card details*, *confirm purchase*, *issue the ticket*). If, for some reason, the services that check the credit card details and the one that issues the ticket would switch place, the composed service would not provide the required functionality. To prevent this from happening, one should have means of checking the correctness of a service composition at run-time.

To address such needs, in Section IV, we propose a hierarchical language for dynamic service composition (HDCL) that allows creating new services, adding, deleting services from lists, as well as connecting services sequentially, or in parallel. We show how to prove the correctness of compositions by checking boolean relations between the involved services' pre-, postconditions. Next, we apply the approach on an abstracted version of an intelligent shuttle system, for which we also compute resource-efficient behaviors, and energy-time trade-offs, by model-checking the system's underlying Priced Timed Automata (recalled in Section II) semantic representation.

In Section VI, we compare to some of the relevant related work, before concluding the paper in Section VII.

II. Preliminaries

A. REMES modeling language

The REsource Model for Embedded Systems REMES [15] is intended as a meaningful basis for modeling and analysis of resource-constrained behavior of embedded systems. REMES provides means for modeling of both continuous (i.e., power) and discrete resources (i.e., memory access to external devices). REMES is a state-machine behavioral language that supports hierarchical modeling, continuous time, and a notion of explicit entry and exit points, making it fit for component-based system modeling.

To enable formal analysis, REMES models can be transformed into timed automata (TA) [1], or priced timed automata (PTA) [2], depending on the analysis type.

The internal component behavior in REMES is given in terms of modes that can be either *atomic* (do not contain submode(s)), or *composite* (contain submode(s)). The data transfer between modes is done through the *data interface*, while the control is passed via the *control interface* (i.e., entry and exit points). REMES assumes *local* or *global* variables that can be of types boolean, natural, integer, array, or clock (continuous variable evolving at rate 1).

Each (sub)mode can be annotated with the corresponding continuous resource usage, if any, modeled by the first derivative of the real-valued variables that denote resources, and which evolve at positive integer rates.

The control flow is given by the set of directed lines (i.e., *edges*) that connect the control points of (sub)modes. REMES supports *delay/timed* actions and *discrete* actions. The former describe the continuous behavior of the mode, and their execution does not change the current mode; the latter, discrete actions (represented as edge annotations), when fired, result in a mode change. The delay/timed actions are not exposed in the model, but are constrained by the above mentioned differential equations. In order for a discrete action to be executed, the corresponding boolean *guard*, which prefixes the action body, must hold. Modes may also be annotated with *invariants*, which bound from above the current mode's delay/execution time. For a more thorough description of the REMES model, we refer the reader to [15].

B. Priced Timed Automata

In the following, we recall the model of priced (or weighted) timed automata [2], [6], an extension of timed automata [1] with prices/costs on both locations and transitions.

Let X be a finite set of clocks and $\mathcal{B}(X)$ the set of formulas obtained as conjunctions of atomic constraints of the form $x \bowtie n$, where $x \in X$, $n \in \mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. The elements of $\mathcal{B}(X)$ are called *clock constraints* over X.

Definition 1: A linearly Priced Timed Automaton (PTA) over clocks X and actions Act is a tuple (L, l_0, E, I, P), where L is a finite set of locations, l_0 is the initial location, $E \subseteq L \times \mathcal{B}(X) \times Act \times \mathcal{P}(X) \times L$ is the set of edges, $I : L \to \mathcal{B}(X)$ assigns invariants to locations, and $P : (L \cup E) \to \mathbb{N}$ assigns prices (or costs) to both locations and edges. In the case of $(l, g, a, r, l') \in E$, we write $l \overset{g,a,r}{\to} l'$. ■

The semantics of a PTA is defined in terms of a timed transition system over states of the form (l, u), where l is a location, $u \in \mathbf{R}^X$, and the initial state is (l_0, u_0), where u_0 assigned all clocks in X to 0. Intuitively, there are two kinds of transitions: delay transitions and discrete transitions. In delay transitions,

$$(l, u) \overset{d,p}{\to} (l, u \oplus d)$$

the assignment $u \oplus d$ is the result obtained by incrementing all clocks of the automata with the delay amount d, and $p = P(l) * d$ is the cost of performing the delay. Discrete transitions

$$(l, u) \overset{a,p}{\to} (l', u')$$

correspond to taking an edge $l \overset{g,a,r}{\to} l'$ for which the guard g is satisfied by u. The clock valuation u' of the target state is obtained by modifying u according to updates r. The cost $p = P((l, g, a, r, l'))$ is the price associated with the edge.

A timed trace σ of a PTA is a sequence of alternating delays and action transitions

$$\sigma = (l_0, u_0) \overset{a_1,p_1}{\to} (l_1, u_1) \overset{a_2,p_2}{\to} \dots \overset{a_n,p_n}{\to} (l_n, u_n)$$

and the cost of performing σ is $\sum_{i=1}^{n} p_i$. For a given state (l, u), the minimum cost of reaching (l, u) is the infimum of the costs of the finite traces ending in (l, u). Dually, the maximum cost of reaching (l, u) is the supremum of the costs of the finite traces ending in (l, u).

A network of PTA $A_1||\ldots||A_n$ over X and Act is defined as the parallel composition of n PTA over X and Act. Semantically, a network again describes a timed transition system obtained from those components, by requiring synchrony on delay transitions and requiring discrete transitions to synchronize on complementary actions (i.e. $a?$ is complementary to $a!$).

III. Behavioral modeling of services in REMES

In REMES, a service is represented by a mode (be it atomic or composite). The service may have a special Init entry point, visited when the service first executes, and where all variables are initialized. In order for a service to be published and later discovered, a list of attributes should be exposed at the interface of a REMES mode/service (see Fig.1).

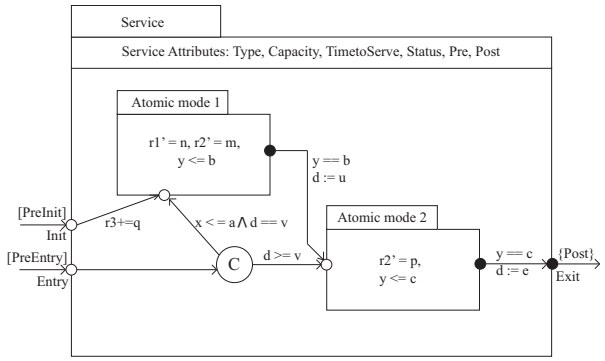


Figure 1. A service modeled in REMES

The attributes depicted in Fig.1 have the following meaning:

- service type - specifies whether the given service is a web service (i.e., weather report), a database service (i.e., ATM services), a network service, etc.;
- service capacity - specifies the service's maximum ability to handle a given number of messages per time unit (i.e., the maximum service frequency)($\in \mathsf{N}$);
- time-to-serve - specifies the worst-case time needed for a service to respond and serve a given request ($\in \mathsf{N}$);
- service status - describes the current service status (that is, passive (not invoked), idle, active);
- service precondition - is a predicate (boolean) ($\mathsf{Pre} = \mathsf{PreInit} \vee \mathsf{PreEntry}$) that conditions the start of service execution, that is, it must be true at the time a REMES service is invoked;
- service postcondition - is a predicate that must hold at the end of a REMES service execution.

When publishing a REMES service, such information is presented as follows:

$$\mathsf{Service} ::= < \mathsf{Type}, (\text{no. of messages / time unit}), \mathsf{Status}, \text{no. of time units}, \mathsf{Pre}, \mathsf{Post} >,$$

where:

$$\begin{array}{lcl} \mathsf{Type} & \in & \{\text{web service}, \text{database}, \text{network service}, \ldots\} \\ \mathsf{Status} & \in & \{\text{passive}, \text{idle}, \text{active}\} \end{array}$$

Such properties are also used to discover Service: the attributes are specified by an interested party and, based on the specification, the service is retrieved or not.

A. Semantics of REMES services

The formal definition of a REMES service is a PTA in which we deliberately replace the function that assigns costs to locations and edges with a function that assigns resource usage values to modes and edges in REMES.

Definition 2: A REMES service over clocks X is a tuple (M, m_0, E, I, Res), where M is a finite set of modes m, m_0 is the initial mode, $E \subseteq M \times \mathcal{B}(X) \times Act \times 2^X \times M$ is the set of edges, $I : M \rightarrow \mathcal{B}(X)$ assigns invariants to modes, and $Res : (M \cup E) \rightarrow \mathbb{N}$ assigns resource-usage values to both modes and edges. ■

We denote by Act the set of actions that assign values to clocks, and other variables. The semantics of a REMES service is given as a *labeled timed transition system with resources* (LTTS), with *discrete transitions* that result in changing the current state, and *delay transitions* that do not change the state but result in time progress. A run of a REMES service is a path in the underlying transition system. Given a run $\xi = s_0 \xrightarrow{r^0} s_1 \xrightarrow{r^1} \ldots \xrightarrow{r^{n-1}} s_n$, where $s_0, \ldots, s_n$ are states, and $r^0, \ldots, r^{n-1}$ are the corresponding resource usage values per transition, respectively, its i^{th} accumulated resource-usage value is $Res_i(\xi) = \sum_{j=0}^{n-1} r_i^j$.

In order to support the dynamic nature of services, we have extended the original REMES [15] with constructs that enable run-time operations on services (e.g. create, delete, compose service, etc.). The description of all operations is given in Section IV.

B. Correctness and interface refinement of REMES services

In proving correctness properties of a single REMES service expressed in terms of a PTA, we resort to the *forward analysis*, which assumes computations of *strongest postconditions* of automata, with respect to a given precondition.

Let us assume the Hoare triple, $\{\mathsf{Pre}\}\,\mathsf{Service}\,\{\mathsf{Post}\}$, $\mathsf{Pre}, \mathsf{Post}$ predicates, denoting the *partial* correctness of Service with respect to precondition Pre and postcondition Post. Introduced by Dijkstra and Scholten [8], the

strongest postcondition predicate transformer (a function that maps predicates to predicates), in our case denoted by sp.Service.Pre, holds in those final states for which there exists a computation controlled by Service, which belongs to the class "initially p". Proving the Hoare triple, that is, the correctness of a REMES service, reduces then to showing that

$$\mathsf{sp.Service.Pre} \Rightarrow \mathsf{Post}$$

holds. In the following, we will use the shortcut notation Service.Pre for the strongest postcondition.

The computation of the strongest postcondition of a PTA is subject to future work. However, we intend to build on the preliminary results of Badban et al. [5], who propose an algorithm (called CIPM) that computes new invariants for timed automata control locations taking their originally defined invariants as well as the constraints imposed by incoming state transitions into account. The algorithm also prunes from the automaton the transitions that can never be taken. We plan to investigate a similar idea for PTA, in order to compute the strongest invariant of a service, and consequently its strongest postcondition.

A service user, but also a developer of services, might need to replace a service with some better quality of service one. It follows that one needs to be able to check whether the new service still delivers the original functions, while having better time-to-serve or resource-usage qualities. Verifying such a property reduces to proving refinement of services.

Assuming two REMES services, $\mathsf{Service}_1$, $\mathsf{Service}_2$, over sets of state variables Σ_1, Σ_2, respectively, we say that $\mathsf{Service}_2$ is a refinement of $\mathsf{Service}_1$, denoted by $\mathsf{Service}_2 \preceq \mathsf{Service}_1$, iff:

$$\{\mathsf{Pre}_2\}\,\mathsf{Service}_2\,\{\mathsf{Post}_2\} \Rightarrow \{\mathsf{Pre}_1\}\,\mathsf{Service}_1\,\{\mathsf{Post}_1\}$$

This definition is consistent with the refinement calculus [4], in the sense that either weakening the precondition or strengthening the postcondition refines a service.

IV. Hierarchical language for dynamic service composition

Service compositions may lead to complex systems of concurrently executing services. An important aspect of such systems is the correctness of their temporal and resource-wise behavior. In the following, we propose an extension to the REMES language, which provides means to define and support creation, deletion, and composition of fine-grained or coarser-grained services, applicable to different domains. We also investigate a formal way of ensuring the correctness of the composition, based on the strongest postcondition semantics of services.

Let us assume that a service, described by a REMES mode, is denoted by $service\_name_i$, $i \in [1..n]$; then, a service list, denoted by *s_list*, is defined as follows:

$$\mathsf{s\_list} \quad ::= \quad \{service\_name_1, ..., service\_name_n\}$$

In order to support run-time service manipulation, we define a set of REMES interface operations presented below. We denote by Σ the set of service states, respectively, that is, the current collection of variable values.

- Create service: *create service_name*

$$\mathsf{create} \quad : \quad Type \times N \times N \times \text{“}passive\text{”} \times (\Sigma \rightarrow bool) \times (\Sigma \rightarrow bool) \rightarrow service\_name$$

- Remove service: *del service_name*

$$\mathsf{del} \quad : \quad service\_name \rightarrow NULL$$

- Create service list: *create s_list*

$$\mathsf{s\_list} \quad = \quad List()$$

- Delete service list: *del s_list*

$$\mathsf{del} \quad : \quad s\_list \rightarrow NULL$$

- Add service to a list: *add service_name s_list*

$$\mathsf{add} \quad : \quad service\_name \times s\_list \rightarrow s\_list \cup \{sevice\_name\}$$

- Remove service from the list: *del service_name s_list*

$$\mathsf{del} \quad : \quad service\_name \times s\_list \rightarrow s\_list - \{service\_name\}$$

- Iterate a service: *while g do service_name od*

$$\mathsf{while\ g\ do\ service\_name\ od} = (\mu X \cdot \mathsf{if}\ g\ \mathsf{THEN}\ service\_name; X\ \mathsf{ELSE}\ skip\ \mathsf{fi})$$

Note that a new service list can be created by using the constructor *list()*, which holds list values of any type. Such a constructor enables the creation of both empty list and also list with some initial value (e.g., $s\_list = List : String(\{\text{“}\mathsf{Shuttle1}\text{”}\})$). Also, adding a service to a list means, in this context, appending that service, that is, adding it at the end of the list. Inserting a service in a list at a specific position, *insert service_name s_list i*, is done by function insert : $service\_name \times s\_list \times \{1, \ldots, s\_list.length\} \rightarrow s\_list \cup \{sevice\_name\}$. Last but not least, the iterative construct (loop as long as boolean g holds) is defined above as the least fixed point of the unfolding function.

Most often, services can be perceived as independent and distributed functional units, which can be composed to form new services. The systems that result out of service

composition have to be designed to fulfill requirements that often evolve continuously and therefore require adaptation of the existing solutions.

Alongside the above operations, we also define a hierarchical language that supports dynamic REMES service composition (HDCL), that is, facilitates modeling of nested sequential, parallel or synchronized services:

$$\begin{array}{lll} \mathsf{DCL} & ::= & (s\_list,\ PROTOCOL,\ REQ\ ) \\ \mathsf{HDCL} & ::= & (((DCL^{+},\ PROTOCOL,\ REQ\ )^{+}, \\ & & PROTOCOL,\ REQ)^{+} \ldots\ ) \end{array}$$

The positive closure operator is used to express that one or more DCLs (dynamic composition languages) are needed to form an HDCL. The *PROTOCOL* defines the way services are composed, that is, the type of binding between services, as follows:

$$\begin{array}{lll} \mathsf{PROTOCOL} & ::= & \mathsf{unary\_operator}\ service\_name \\ & & \mid service_m\ \mathsf{binary\_operator}\ service_n \end{array}$$

The requirement *REQ* is a predicate ($\Sigma \rightarrow \mathsf{Bool}$) that can include both functional and extra-functional properties/constraints of the composition. It identifies the required attribute constraints, capability, characteristics, or quality of a system, such that it exhibits the value and utility requested by the user. The above unary and binary operators are defined as follows:

$$\begin{array}{lllllllll} \mathsf{Unary\_operator} & ::= & \mathsf{exec-first} & & & & & & \\ \mathsf{Binary\_operator} & ::= & ; & \mid & \| & \mid & \|_{SYNC} & & \end{array}$$

Let us assume that two services s_1, s_2 are invoked at some point in time, and their instances are placed in the service list s_{list}. Also, we assume that $s_i.Pre_i$ is the strongest postcondition of s_i, $i \in 1, 2$, w.r.t. precondition Pre_i. Then, the semantics of the unary and binary protocol operators, as well as the correctness conditions for such compositions are given as follows.

- Exec-first (specifies which service should be initially executed in a composition) - below we formalize the fact that s_1 should execute first, and only when it finishes and establishes its postcondition, service s_2 can become active:

$$\begin{array}{l} status_{s_1} == active \\ \wedge\ status_{s_2} == idle \\ \wedge\ Post_{s_1} \Rightarrow (status_{s_2} == active) \end{array}$$

- Sequential composition - first, we give the semantics of this composition, after which we show the correctness condition:

$$\begin{array}{lcl} (s_1\ ;\ s_2).Pre_{s_1} & == & s_2.(s_1.Pre_{s_1}) \\ (s_2.(s_1.Pre_{s_1}) \Rightarrow Post_{s_2}) & \wedge & (Post_{s_2} \Rightarrow REQ) \end{array}$$

- Parallel composition's $(s_1 \parallel s_2)$ correctness condition:

$$(s_1.Pre_{s_1} \vee s_2.Pre_{s_2}) \Rightarrow REQ$$

- Parallel composition with synchronization - we denote by SYNC the set of preconditions of services that need to synchronize their executions:

$$\begin{array}{l} (s_1 \parallel_{SYNC} s_2) \\ \triangleq (Pre_{s_1}, Pre_{s_2} \in \mathsf{SYNC} \Rightarrow (status_{s_1} == status_{s_2} == active)) \end{array}$$

Modeling the actual mechanism of synchronizing REMES modes is subject to future work. However, the usefulness of the language is demonstrated by the following case-study.

V. Example: An Autonomous shuttle system

We consider a simplified version of [10] developed at University of Paderborn within the Railcab project, to demonstrate behavioral modeling and composition of services in REMES. The aim of project is to develop an intelligent, flexible, cost and resource effective rail-based transportation system which can be scheduled on-demand. In our example we extract parts of behavior described in [10] to show how services are created, invoked, composed, and idled. We consider a system of three trains that provides service of transportation to three different locations. Each of the trains have an in advance defined path to be followed as depicted in Fig. 2. During the transport shuttles might meet at point B in Fig. 2 in which they are forced to create a convoy. In order to enter the convoy they have to respect given speed and acceleration limit measured in points A1, A2, and A3 as depicted in Fig. 2, otherwise they may stop to let others that fulfil the given requirements join the convoy. After a convoy is formed and has left, those that were stopped are allowed to continue their journey to previously assigned destination if sensor in point C in Fig. 2 have sent signal that it is safe to continue (i.e., the distance from the formed convoy is long enough to avoid possible collisions). Each stop and start increases consumption of power, and possibility to miss the deadline that brings penalty in terms of increased cost. The overall cost is accumulated based on resource consumption and penalty or reward. We assume that one of the operating shuttles is older than other two, i.e., consumes more of the available resources to fulfill given requests.

After the destination point is being reached, shuttle is free to go to idle state and wait for a new order. Above described system is equipped with one central controller as shown in Fig. 2 that based on a service description provided with each shuttle decides when and which shuttle to invoke.

A. Modeling the shuttle system in REMES

We model the internal behavior of the Autonomous shuttle system services as modes in REMES. The composite

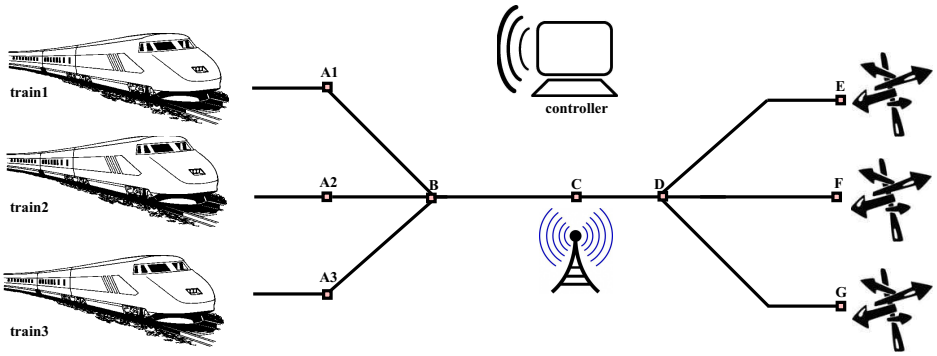


Figure 2. An example overview.

modes of Shuttle1 and Controller1 are depicted in Fig. 3 and Fig. 4 respectively. They consist of the *atomic* modes (i.e., Acceleration1, STOP, Destination, etc.), *conditional connectors* (C), and *discrete actions* (e.g., status1:= ready). The modes communicate data between each other using the global variables: speed_i, status_i, t_i, and StatusConvoy. Control interfaces are used to expose mode attributes relevant for mode discovery.

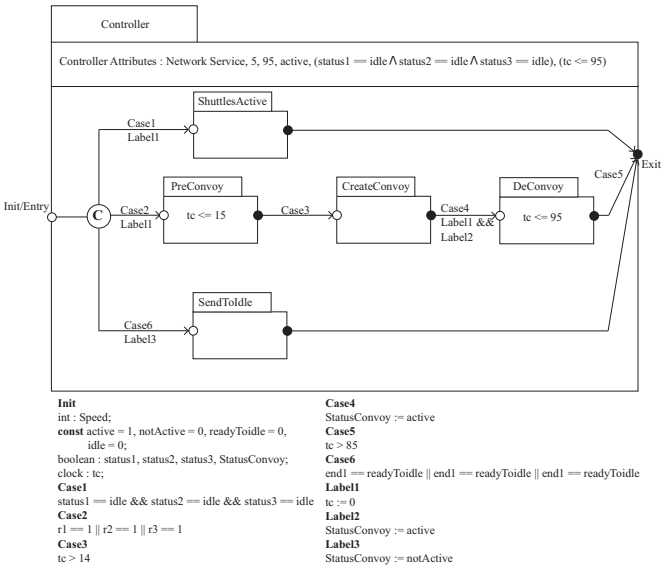


Figure 4. The model of a controller given as a REMES mode

Shuttle1 and Shuttle3 have the same behavior. Shuttle2 is an older shuttle than the other two, therefore it requires more time to start, accelerate, slow down, and its resource consumption is slightly higher than for the other two. Due to the space limitation, we will not show and describe in detail all modes, only of Shuttle1.

When Shuttle1 is activated, the shuttle starts to accelerate in submode Acceleration1. In each run, the shuttle accelerates by 10 speed units in time $\text{t}_1 \in [7, 10]$. The acceleration continues up to 70 speed units, after which the shuttle can either enter submode Convoy , or continue to accelerate. If the shuttle joins a convoy together with other shuttles it will start to accelerate until it reaches its full speed, (speed_1 == 120). It will then start to slow down in order to deattach from the convoy. The acceleration, deacceleration, and deconvoying are done in the modes: CAcceleration, CSlowDown, and DeConvoy, respectively. If a shuttle fails to join a convoy, it is stopped in submode STOP until it is safe to continue alone. When the signal StatusConvoy is reset to notActive, a stopped shuttle restarts and continues towards its destination point. In the destination point, a shuttle can be rewarded or punished (the predefined penalty in terms of cost is assigned), depending on if the deadline is met or not.

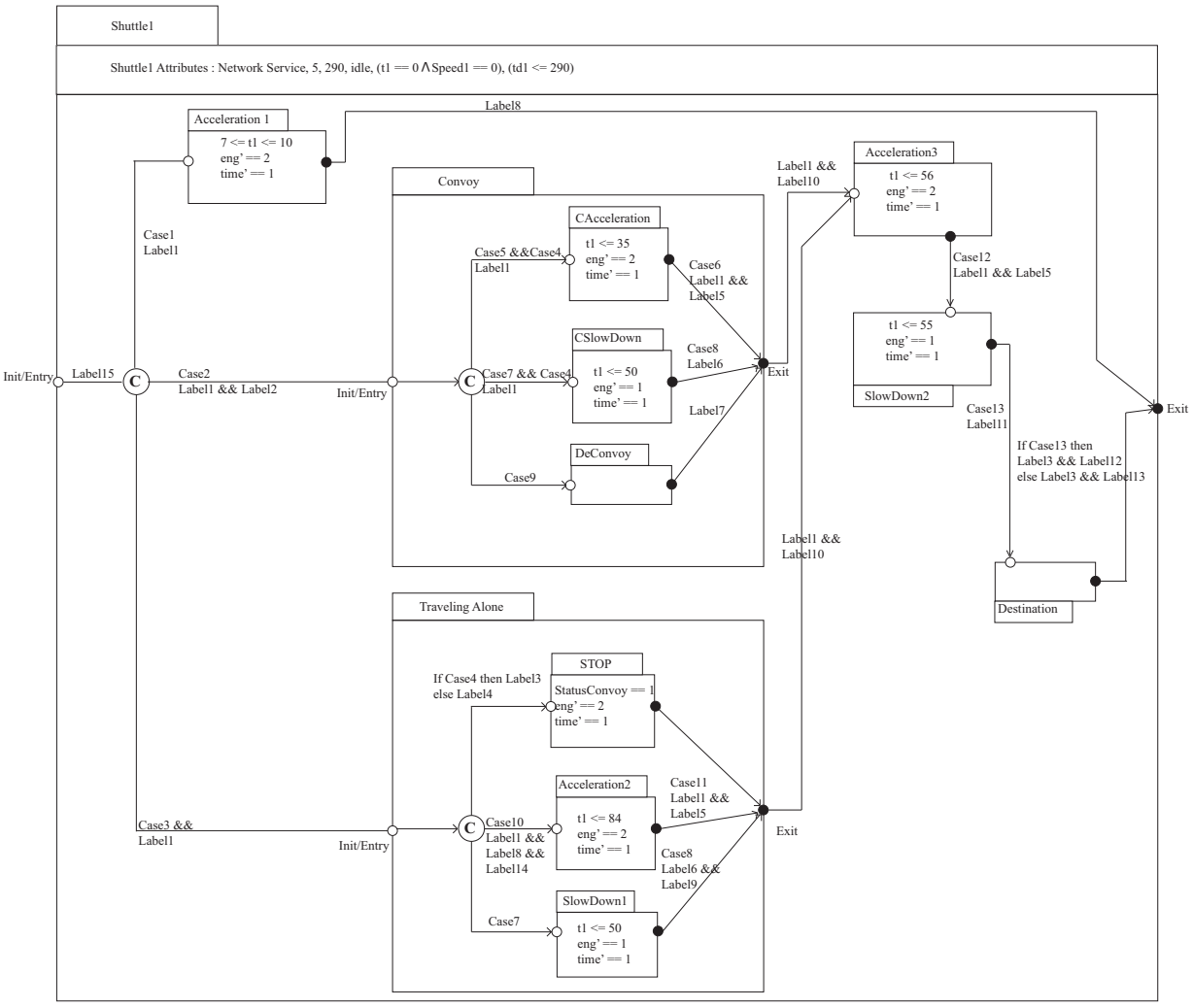
The Controller mode sends an activation signal if all shuttles are ready to start. For those shuttles that fulfill the speed limitation, the Controller mode enables entrance to the convoy. Later, the mode will deconvoy shuttles and, when they reach final destination, the Controller mode sends them to the idle state. Execution of actions in Shuttle1 and Controller modes is controlled by the clock variables t_1 and t_c.

In the example, we make use of two resources: power and time. Our assumption is that each acceleration and slow down utilizes certain amount of power and time. The consumption is increased if shuttles are forced to stop and then restart again. In order to carry out analysis, REMES-based Autonomous shuttle system is translated to a network of PTA. For more details regarding REMES models, we refer the reader to [15].

B. Applying the hierarchical language

To illustrate the use of the hierarchical language for modeling service composition, depicted in Fig. 5, we recall the example description of Section V.

Through the declarative part, the needed services are introduced (lines 00-17 in Fig. 5). A service declaration



Init
const ready = 1, active = 1, notActive = 0,
readyToidle =1, idle = 0
int : speed1, r1;
boolean : StatusConvoy, status1, alone1,alone2
alone3;
clock : t1, td1;
resource eng : T_ϵ ;
resource time : T_ϵ ;
Case1
status1 == ready && speed1 <= 80
Case2
speed1 == 70 && StatusConvoy == notActive && t1 == 0
Case3
speed1 > 70
Case4
StatusConvoy ==active
Case5
speed1 < 120
Case6
t1 == 35
Case7
speed1 == 120
Case8
t1 ==50
Case9
speed == 15
Case10
StatusConvoy == notActive && alone2 ==0 && alone 3 == 0
Case 11
t1 == 84
Case12
t1 == 56
Case13
t1 == 55 && end1 == readyToidle
Case14
t1d <= 290
Label1
t1:=0
Label2
r1 :=1
Label3
speed1 := 0
Label4
speed1 := speed1
Label5
speed1 := 120
Lablel6
speed1 := 15
Label7
r1 := 0
Label8
speed1 +=10
Label9
alone1 := 0
Label10
speed1 += 20
Label11
speed1 := 5
Label12
status1 := idle, end1 := idle, cost += 5
Label13
status1 := idle, end1 := idle, cost += 10
Label14
alone1 := 1
Label15
status1 := ready

Figure 3. The model of a shuttle1 given as a REMES service

contains service name, type, status, TimeToServe, precondition and postcondition. The corresponding requirement is matched against such attribute information, when choosing a service. After the selection, the instances of the selected services are created (lines 18-20 in Fig. 5), and added to the service list using the *add* command (lines 22-23 in Fig. 5). Finally, the chosen services are composed by DCL. The list of services, employed protocol (type of service binding), and DCL requirements are given as parameters. Moreover, the language provides means to compose the existing DCLs with other services, thorugh HDCL, as shown in line 25 of Fig. 5. If not anymore needed, the

composition can be deleted.

The advantage of this language is that, after each composition, one can check whether the given requirement is satisfied. The intention of this example is to show how the language syntax looks, since the language does not have a tool support yet. Our intention is to eventually provide both user and developer with an automated way of checking the services and their compositions against given requirements. The formalization of check conditions is intended to be completely hidden from the user.

```
00 declare Shuttle1 ::= <network service,
01                       5,
02                       290,
03                       idle,
04                       (t1 == 0 ∧ speed ==0),
05                       (t1<=290)>
06 declare Shuttle2 ::= <network service,
07                       7,
08                       300,
09                       idle,
10                       (t2 == 0 ∧ speed ==0),
11                       (t2<=300)>
12 declare Shuttle3 ::= <network service,
13                       5,
14                       290,
15                       idle,
16                       (t3 == 0 ∧ speed ==0),
17                       (t3<=290)>
18 create Shuttle1
19 create Shuttle2
20 create Shuttle3
21 create list_Convoy
22 add Shuttle1 list_Convoy
23 add Shuttle2 list_Convoy
24 DCL_Convoy ::= (list_Convoy, ; , t<= 300)
25 HDCL_Convoy ::= ((DCL_Convoy, Shuttle3), | |, t<= 300)
26 check ((Shuttle2.(Shuttle1.(t1 == 0 ∧ speed == 0)))
          ∧ (t == t1 ∨ t == t2)) => (t <= 300)
27 check ((Shuttle3.(t3 == 0 ∧ speed == 0)) ∧ (t == t3))
         => (t <= 300)
28 del HDCL_Convoy
```

Figure 5. An illustration of the REMES language

C. A PTA model of the shuttle system

We have analyzed the Autonomous shuttle system as a network of five PTA in UPPAAL CORA[1]. The automata, denoted as RailCab1, RailCab2, and RailCab3, are offering services of transportation to predefined destinations, and are being controlled by the PTA CentralController1 and CentralController2. We have chosen to split the controller function into two parallel timed automata, where one automata activates shuttles and synchronizes them into a convoy, and the other idles shuttles when they reach the final destination. The models of RailCab1, CentralController1 and CentralController2 are shown in Fig. 6, Fig. 7(a), and Fig. 7(b). Due to space limitations, the other two automata (which are similar to RailCab1) are not shown.

The shuttles are modeled as PTA with locations: Idle, Acceleration1, Convoy, ConvoyAcceleration, ConvoySlowDown, DeConvoy, SeparationPoint, STOP, Acceleration2, SlowDown1, Acceleration3, SlowDown2, and DestinationReached. When all shuttles are in an Idle locations, they receive a synchronization signal through the broadcast channel activateRC from CentralController1 that activates them. At the same time, boolean variable status_i is set to true, indicating that the shuttle is active. The acceleration is performed through several iterations in location Acceleration1. In each iteration, the shuttles accelerate 10 speed units within the time bounds $7 \leq t_i \leq 10$. The speed information is kept in the bound integer variable RCspeed_i. Shuttles that are at the same time in location Acceleration1, and have exactly 70 in speed, are allowed to form a convoy and continue together until the SeparationPoint. The boolean variable r_i is true when a shuttle joins a convoy and reset to false whenever it leaves a convoy. While being in a convoy, shuttles are controlled and synchronized by CentralController1 through the two broadcast channels SynConvoy1 and SynConvoy2. After leaving a convoy in location SeparationPoint, each shuttle is continuing towards its predefined destination point. Synchronization channels idle1, idle2, and idle3 are sent to CentralController2 when shuttles are ready to go to the idle state.

A shuttle that has speed greater than 70 must proceed alone towards the destination point. Before continuing, the function ConvoyCheck_i(StatusConvoy) is used to reset the speed of the shuttle (to enable stopping in location STOP) and to check whether a convoy exists at that moment (boolean variable StatusConvoy set to 1 indicates that a convoy exists on the track). Variable StatusConvoy is used to avoid the possible collision between the convoy and the remaining shuttle(s). In case the track is empty, a shuttle is allowed to continue alone without stopping.

Before entering location Idle, it is checked whether a shuttle has reached DeestinationReached point or not, within the given deadline. The clock variable td_i is used to keep track of time duration of reaching the final destination. Depending on the value of td_i, a shuttle can receive a reward (i.e., destination reached within given deadline) or penalties (i.e., the deadline is missed).

Recall that RailCab2 is assumed to be an older shuttle.

[1]For more information about the UPPAAL CORA tool, visit the web page www.uppaal.org/cora.

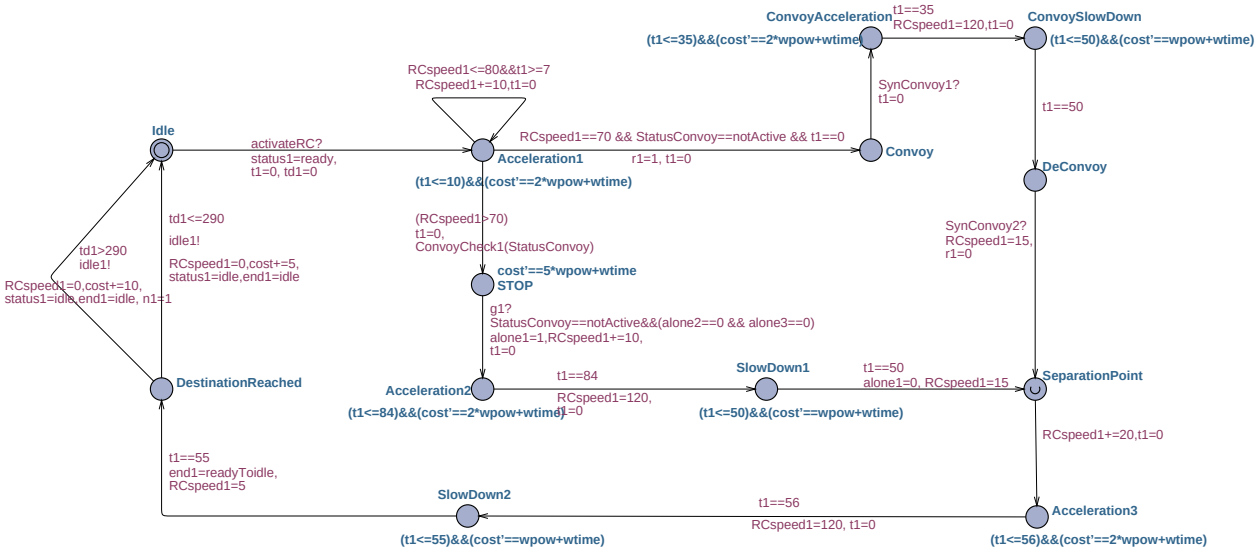


Figure 6. The model of a shuttle given as a PTA

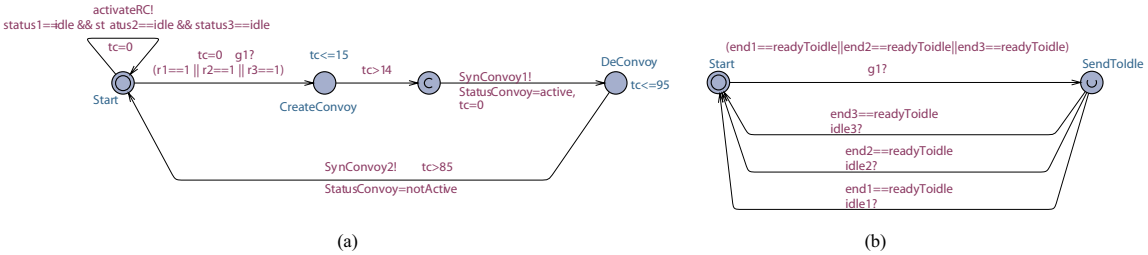


Figure 7. The model of a controller given as two parallel PTAs

The model of this shuttle differs from other two in terms of time needed to accelerate/slow down, and resource consumption while being active.

The PTA of CentralController1 has four locations: Start, PreConvoy, CreateConvoy, and DeConvoy. When all shuttles are in their idle states, CentralController1 activates them by broadcasting signal activateRC. Whenever any of the shuttles sets the boolean variable r_i to 1, CentralController moves to PreConvoy location, where it stays for 15 time units, to allow for other shuttle(s) to fulfill the speed requirement and join the convoy. The broadcast synchronization channel SynConvoy1 is used to enable convoy creation. In DeConvoy location, the controller stays 86 to 95 time units to wait for the shuttles in the convoy to complete the decomposition. In order to deconvoy them, the broadcast synchronization channel SynConvoy2 is sent, and boolean variable StatusConvoy is reset to not active.

The PTA of CentralController2 is responsible for sending the shuttles to their idle state whenever they have reached their destination point (e.g., end_i == readyToidle). The synchronization channels idle_i are used between the shuttles and the CentralController2.

D. Formal analysis of the PTA model

We consider power to be the most critical resource in our system, since each shuttle operates on batteries with limited capacity. We have also taken timing properties into consideration since each shuttle has a predefined deadline to meet. The total cost of resource consumption is further influenced by the individual weight of the resource and the consumed resource on the transitions and locations. In our example, the total cost function is defined as:

$$c_{tot} = \mathsf{wpow} \times c_{pow} + \mathsf{wtime} \times c_{time} \qquad (1)$$

where wpow = 3, wtime =1, and c_{pow} and c_{time} are the accumulated consumed amounts of power and time, respectively.

Using UPPAAL CORA we were able to analyze the minimum cost reachability problem, that is, to compute the lowest cost of satisfying a given reachability property, and a witness trace. During our analysis, we start with validating the system by checking that a shuttle that starts to accelerate is guaranteed to reach its final destination.

$$\mathsf{AG}\ (\mathsf{Shuttle_i.Acceleration1} \implies \mathsf{AF}\ (\mathsf{Shuttle_i.DestinationReached}))$$

We also check that the system is deadlock free AG not deadlock, implying that no shuttle will ever be blocked by any other shuttle(s), in its attempt to reach the final destination. In terms of safety properties, we check that no shuttle will enter the convoy with speed other than 70 speed units, and that a shuttle that is stopped due to not fulfilling the speed limit will never start before it gets the signal to proceed. To illustrate the technique we specify the above safety properties in the UPPAAL property specification language — a subset of Timed Computational Tree Logic (TCTL).

$$\mathsf{AG\ not}(\mathsf{Shuttle_i.speed} > 70 \ \mathsf{and}\ \mathsf{Shuttle_i.Convoy})$$
$$\mathsf{AG}\ ((\mathsf{Convoy} == 1) \Rightarrow \mathsf{Shuttle_i.STOP})$$

where variable $\mathsf{Shuttle}_i$.speed represents the speed of shuttles i $\in \{1,2,3\}$, $\mathsf{Shuttle}_i$.Convoy and $\mathsf{Shuttle}_i$.STOP (convoy entry point and shuttle stop point, respectively) denote locations that the shuttle i $\in \{1,2,3\}$ visits while operating on tracks.

We check the cost of the system based on various scenarios, e.g., all shuttles join a convoy, only two are in convoy, or all shuttles operate on their own, separately. Additionally, we check cases in which shuttles miss the predefined deadline, respectively. Table I shows the cost for

Scenario	S1	S2	S3	Shuttles missed the deadline	$Cost_{st}$	$Cost_{bt}$
1.	+	+	+	0	5107	5107
2.	+	-	+	1	5312	5122
3.	+	+	-	1	5504	5259
	-	+	+	1		
4.	-	+	-	2	5901	5411
5.	+	-	-	2	6709	5474
	-	-	+			
6.	-	-	-	3	7106	5626

Table I. Cost for different shuttle interaction scenarios

different shuttle (S1, S2, S3) interaction scenarios. The cost is given for an arbitrary trace (some trace, $Cost_{st}$), and also for the best trace ($Cost_{bt}$) with the minimum possible system cost. We have identified six different scenarios. The best scenario, the most resource saving scenario is the case in which all shuttles join the convoy and decrease the amount of resources utilized. The fact that a shuttle joins the convoy implies that the overall time will be kept within the given bound, and the shuttle will meet its deadline. The second best case is when two shuttles with the same abilities and resource consumptions join a convoy and meet their respective deadlines, while the third shuttle has to wait and miss its deadline. The overall system cost is increased due to obligatory stop and restart for shuttles that do not join the convoy. As expected, the cost increases with the increase in the number of shuttles that do not join the convoy, since additional time is consumed and given deadlines are missed. The highest cost is obtained in case all shuttles overspeed or underspeed, and do not meet the requirements to join the convoy. The main additional cost increase in this case is the penalty that each shuttle receives because of the missed deadline.

VI. Discussion and related work

Based on the level of details that are provided through the behavioral description, all approaches related to services and SOS can be in principle divided into three groups.

Code-level behavioral description approaches are mostly based on XML language (e.g., BPEL, WS-CDL). BPEL [3] is an orchestration language whose behavioral description includes a sequence of project activities, correlation of messages and process instances, and recovery behavior in case of failures and exceptional conditions. Approaches like BPEL are useful when services are intended to serve a particular model or when the access to the service implementation exists. The drawback of such approaches is the lack of formal analysis support, which forces the designer/developer to master not only the specification and modeling processes, but also the techniques for translating models into a suitable analysis environment.

When compared to the above group, BPMN [12] can be seen as a higher-level language. It relies on a process-oriented approach, and supports a graphical representation to be used by both designers and analysts. The lack of a formal behavioral description makes it not suitable for a detailed analysis as the ones supported by REMES.

The third group includes approaches with formal background. Rychlý describes the service behavior as a component-based system for dynamic architectures [14]. The specification of services, their behavior, and hierarchical composition are formalized within the π-calculus. Similar to our approach, this work emphasizes the behavior in terms of interfaces, (sub)service communication, and bindings, while we also cater for service descriptions

including timing and resource annotations. Foster et al. present an approach for modeling and analysis of web service compositions [9]. The approach takes BPEL4WS service specification, and translates it into Finite State Processes (FSP), and Labeled Transition Systems (LTS), for analysis purposes. The authors argue that the resource constraints of a system should not be omitted, but they are mainly dealing with safety and liveness violation, due to resource constraints, whereas we additionally focus on optimal resource consumption and trade-off analysis.

VII. Conclusions

In this paper, we have presented an approach for formal service description by extending the resource-aware timed behavioral language REMES. Attributes such as type, time-to-serve, capacity, etc., together with precondition and postcondition are added to REMES to enable service discovery, as well as service interaction. We have chosen to use Hoare triples and a strongest postcondition semantics to prove service correctness. We have also proposed a hierarchical language for service composition, which allows for the verification of, e.g., service composition correctness. The approach is demonstrated on a simplified version of an intelligent shuttle system, for which we have computed resource consumptions, and shown energy-time trade-off analysis. As a first partial validation of the proposed approach, we find the results of the case study very encouraging. As future work, we will look into the algorithmic computation of strongest postconditions of priced timed automata, inspired by preliminary results for strongest postcondition computation for ordinary timed automata. Also, we intend to model and formalize synchronization of REMES modes.

References

[1] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: citeseer.nj.nec.com/alur94theory.html

[2] R. Alur, "Optimal paths in weighted timed automata," in *In HSCC01: Hybrid Systems: Computation and Control*. Springer, 2001, pp. 49–62.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, *BPEL4WS, Business Process Execution Language for Web Services Version 1.1*, IBM, 2003.

[4] R. J. R. Back and J. von Wright, *Refinement Calculus: A Systematic Introduction*. Springer–Verlag, 1998.

[5] B. Badban, S. Leue, and J.-G. Smaus, "Automated predicate abstraction for real-time models," *EPTCS*, vol. 10, p. 36, 2009. [Online]. Available: doi:10.4204/EPTCS.10.3

[6] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager, "Minimum-Cost Reachability for Priced Timed Automata," in *Proceedings of the 4th International Workshop on Hybris Systems: Computation and Control*, ser. Lecture Notes in Computer Sciences, M. D. D. Benedetto and A. Sangiovanni-Vincentelli, Eds., no. 2034. Springer–Verlag, 2001, pp. 147–161.

[7] A. Causevic and A. Vulgarakis, "Towards a unified behavioral model for component-based and service-oriented systems," in *2nd IEEE International Workshop on Component-Based Design of Resource-Constrained Systems (CORCS 2009)*. IEEE Computer Society Press, July 2009.

[8] E. W. Dijkstra and C. S. Scholten, *Predicate calculus and program semantics*. New York, NY, USA: Springer-Verlag New York, Inc., 1990.

[9] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. Rosenblum, and S. Uchitel, "Model checking service compositions under resource constraints," in *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 225–234.

[10] H. Giese and F. Klein, "Autonomous shuttle system case study," in *Scenarios: Models, Transformations and Tools*, 2003, pp. 90–94.

[11] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto, "Web services choreography description language version 1.0," World Wide Web Consortium, Candidate Recommendation CR-ws-cdl-10-20051109, November 2005.

[12] *Business Process Modeling Notation (BPMN) version 1.1.*, Object Management Group (OMG), January 2008. [Online]. Available: http://www.omg.org/spec/BPMN/1.1/

[13] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel, "Web service modeling ontology," *Applied Ontology*, vol. 1, no. 1, pp. 77–106, 2005.

[14] M. Rychl, "Behavioural modeling of services: from service-oriented architecture to component-based system," in *Software Engineering Techniques in Progress*. Wroclaw University of Technology, 2008, pp. 13–27.

[15] C. Seceleanu, A. Vulgarakis, and P. Pettersson, "Remes: A resource model for embedded systems," in *In Proc. of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009)*. IEEE Computer Society, June 2009.