Modeling and Reasoning about Service Behaviors and their Compositions

Aida Čaušević, Cristina Seceleanu, and 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 need 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.

1 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, 12, 14, 15] to ensure QoS, but usually it is not straightforward to work out the exact formal analysis model.

In order to fully understand the ways in which services evolve and impact on QoS attributes, a *service behavioral description* is required [6]. Such behavior is assumed to be internal to the service, and hidden from the user. It should include the representation of a service functionality, enabled actions, resource annotations, and possible interactions with other services.

To meet the above demands, in this paper, concretely in Section 3, we extend the existing resource-aware, timed hierarchical language REMES [19], recalled in Section 2, to become 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. Still in Section 3, we introduce a synchronization mechanism for REMES modes, which enables modeling and verification of synchronized services.

By exploiting the pre-, postcondition annotations, we show how to describe the service behavior in Dijkstra's guarded command language [8], and how to check the service correctness by employing Dijkstra's and Scholten's strongest postcondition semantics [9].

Since services can be composed at run-time, analyzing the correctness of a service in isolation does not suffice. To exemplify, let us consider a service that is composed of several navigation services, out of which some return the route length in miles, whereas others in kilometers. If the developer has omitted to introduce a service that converts length from one metric to the other, it is desirable to uncover such an error right away, by formally checking the correctness of the actual service composition, at run-time.

To address the dynamic aspects of services, in Section 4, we propose a hierarchical language for dynamic service composition (HDCL) that allows creating new services, via binary operators, as well as adding and/or deleting services from lists. In the same section, we also give the semantics of sequential, parallel, and parallel with synchronization service composition, respectively. Next, we apply the approach on an abstracted version of an intelligent shuttle system, for which we show the use of REMES language to model the system and apply HDCL language to check the correctness of service compositions. In Section 6, we compare to some of the relevant related work, before concluding the paper in Section 7.

2 Preliminaries

2.1 REMES modeling language

The REsource Model for Embedded Systems REMES [19] 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. 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 [19].

The REMES language benefits from a set of tools[1] for modeling, simulation and transformation into PTA, which could assist the designer during system development.

2.2 Guarded command language

The Guarded Command Language (GCL) was introduced and defined by Dijkstra for predicate transformers semantics [8]. The basic element of the language is the guarded command, a statement list prefixed by a boolean expression, which can be executed only when the boolean expression is initially true.

The syntax of the GCL is given in Backus-Naur Form (BNF) extended with braces "$\{..\}$", where the braces mean: "followed by zero or more instances of the enclosed".

$$\begin{array}{ll}
<\text{guarded command}> & ::= <\text{guard}> \; - > <\text{guarded list}> \\
<\text{guard}> & ::= <\text{boolean expression}> \\
<\text{guarded list}> & ::= <\text{statement}> \{; <\text{statement}>\} \\
<\text{guarded command set}> & ::= <\text{guarded command}> \{ \; [\!] \; <\text{guarded command}>\} \\
<\text{alternative construct}> & ::= \mathbf{if} \; <\text{guarded command set}> \; \mathbf{fi} \\
<\text{statement}> & ::= <\text{alternative construct}> \mid \text{“other statements''} \\
<\text{repetitive construct}> & ::= \mathbf{do} <\text{guarded command set}> \mathbf{od}
\end{array}$$

The semicolons in the guarded list denote that whenever the guarded list is selected for execution, its statements will be executed successively in the order from the left to the right. A guarded command is not a statement but a component of a guarded command set from which statements can be constructed. The separator " $[\!]$ " is used for mutual separation of guarded commands in guarded command set.

The alternative construct is written using special bracket pair: "**if ... fi**". The program aborts if none of the guards is true, otherwise an arbitrary guarded list with a true guard will be executed. Similarly, the repetitive construct "do ... od" means that the program runs as long as one of the guards is true, and terminates if none of the guards is true.

Semantics and Correctness of Guarded Commands. Let us assume the Hoare triple, $\{p\}\ S\ \{q\}$, where p, q are predicates, denoting the partial correctness of guarded command S with respect to precondition p and postcondition q. Introduced by Dijkstra and Sholten [9], the *strongest postcondition predicate transformer* (a function that maps predicates to predicates), denoted by $sp.S.p$, holds in those final states for which there exists a computation controlled by S, which belongs to class "initially p". Proving the Hoare triple, that is, the correctness of a guarded command, reduces to showing that $(sp.S.p \Rightarrow q)$ holds. The strongest postcondition rules for the assignment statement, for sequential composition, and for the non-deterministic conditional are as follows:

$$sp.(x := e).p(x) \equiv x = e \wedge (\exists x \cdot p(x)) \tag{1}$$

$$sp.(S_1; S_2).p \equiv sp.S_2.(sp.S_1.p), \forall p \tag{2}$$

$$sp.(\text{if } g_1 \rightarrow S_1 \; [\!] \; \ldots \; [\!] \; g_n \rightarrow S_n \text{ fi}).p \equiv sp.S_1.(g_1 \wedge p) \vee \ldots \vee sp.S_n.(g_n \wedge p), \forall p \tag{3}$$

[1] The REMES tool-chain is available at http://www.fer.hr/dices/remes-ide.

3 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).

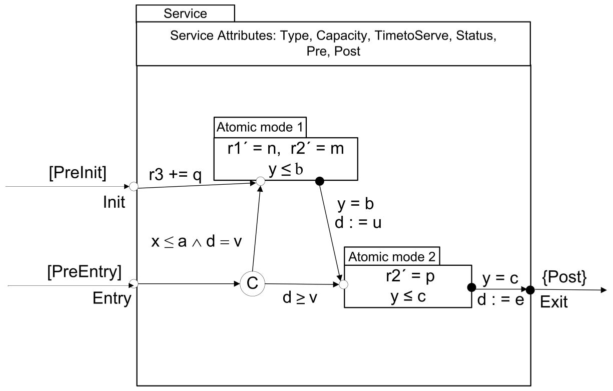

Fig. 1. A service modeled in REMES

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 ($\mathsf{Pre} : \sum \rightarrow \mathsf{Bool}$, $\mathsf{Pre} \equiv (\mathsf{PreInit} \vee \mathsf{PreEntry})$) that conditions the start of service execution, and must be true at the time a REMES service is invoked. In this expression $\sum$ is the polymorphic type of the state that includes both local and global variables, and predicates PreInit, PreEntry are the initial, and the entry precondition of the service, respectively;
– service postcondition - is a predicate (Post) that must hold at the end of a REMES service execution.

The attributes are used to discover Service; they are specified by an interested party and, based on the specification, the service is either retrieved or not.

The formal specification of a service, modeled as the composite mode of Fig. 1, is the Hoare triple $\{p\}\mathsf{Service}\{q\}$, where Service is described in terms of the guarded command language, and the mode's precondition p, and postcondition (requirement) q are as follows:

$$p \equiv y \leq c \wedge c > b \wedge (d = 0 \vee v \leq d \leq e) \wedge r1 = r2 = r3 = 0 \wedge (h = 0 \vee h = 1)$$

$$q \equiv y \leq c \wedge d \leq e \wedge (\forall i, 1 \leq i \leq 3 \cdot ri \leq val_i)$$

where val_i are the given upper bounds on each resource usage, respectively.

Below, we give the GCL description of the REMES composite mode Service:

Service ::=

```
IF
$\neg u1 \wedge h = 0 \wedge y \leq b$                                          Init → Atomic mode 1
  $\rightarrow r3 := r3 + q;$
     $sm :=$ Atomic mode 1; $u1 :=$ true;
     Update(now)
$[\!]$ $\neg u2 \wedge h = 1 \wedge (x \leq a \wedge d = v) \wedge y \leq b$           Entry → Atomic mode 1
  $\rightarrow sm :=$ Atomic mode 2; $u2 :=$ true;
     Update(now)
$[\!]$ $(\neg u3 \wedge (h = 1 \wedge d \geq v) \vee d = u) \wedge y \leq c$   (Entry or Atomic mode 1) → Atomic mode 2
  $\rightarrow sm :=$ Atomic mode 2; $u3 :=$ true;
     Update(now)
$[\!]$ $\neg u4 \wedge sm =$ Atomic mode 1 $\wedge y \leq b$                      Delay in Atomic mode 1
  $\rightarrow r1(t) := r1(now) + n * (t - now);$
     $r2(t) := r2(now) + m * (t - now);$
     $\{y \leq b\}; u4 :=$ true;
     Update(now)
$[\!]$ $\neg u5 \wedge sm =$ Atomic mode 1 $\wedge y = b$
  $\rightarrow d := u; u5 :=$ true;
     Update(now)
$[\!]$ $\neg u6 \wedge sm =$ Atomic mode 2 $\wedge y \leq c$                      Delay in Atomic mode 2
  $\rightarrow r2(t) := r2(now) + p * (t - now);$
     $\{y \leq c\}; u6 :=$ true;
     Update(now)
$[\!]$ $\neg u7 \wedge sm =$ Atomic mode 2 $\wedge y = c$                        Atomic mode 2 → Exit
  $\rightarrow d := e;$
     $h := 1; u7 :=$ true;
     Update(now); $u1, \ldots, u7 :=$ false
FI
```

(4)

In the GCL description (4), the variables x, y are clocks, h is the history variable that is used to decide where to enter the composite mode, sm is the variable ranging over submodes, and $r1 : \mathsf{Real}_+ \rightarrow \mathsf{T}_1$, $r2 : \mathsf{Real}_+ \rightarrow \mathsf{T}_2$ are the continuous resources of the model, defined as functions over the non-negative reals that are used as the time domain. In addition, u_i are local variables used for preventing executing the same action more than once, at the same time point. These variables are reset each time the mode Service exits. Similar to the approach taken for action system models [18], the variable

now shows the current time, and it is explicitly updated by statement $\mathsf{Update}(\mathsf{now})$. The assertions $\{y \leq b\}, \{y \leq c\}$ model the invariants (Inv) of Atomic mode 1, and Atomic mode 2, respectively.

We define $\mathsf{Update}(\mathsf{now})$ as follows:

$$Update(now) \triangleq now := \mathsf{next}.now$$

The submodes can be urgent (no delays are allowed), or non-urgent (where delays can happen, until an invariant Inv is violated); also, guarded actions can annotate edges connecting the entry points of the composite mode with submodes, via some conditional connector (denoted by encircled C in Figure 1). Given these, and assuming that gg is the disjunction of the action guards of the edges leaving a mode (or a conditional connector), and that Inv is the invariant of the respective mode, next is defined by:

$$\mathsf{next}.t \triangleq \begin{cases} min\{t' \geq t \ \mid \ \neg Inv \vee gg\}, & \text{if exists } t' \geq t \text{ such that } \neg Inv \vee gg \\ +\infty, & \text{otherwise.} \end{cases}$$

If a mode is urgent, or the guards correspond to a conditional connector, then $I \equiv \mathsf{false}$, so the next moment of time is identical to the current one, no delay being possible.

The mode Service, modeled by (4), can be iterated for as long as needed, so the complete specification is: $\mathsf{status}_{\mathsf{Service}} := \mathsf{active}; (\mathsf{DO}\ \mathsf{g} \rightarrow \mathsf{Service}\ [\!]\ \neg\mathsf{g} \rightarrow \mathsf{status}_{\mathsf{Service}} := \mathsf{idle}\ \mathsf{OD})$. According to rule (3), the strongest postcondition of the conditional statement is:

$$\begin{array}{ll} & sp.Service.p \\ \equiv & \\ & sp.(r3 := r3 + q; sm := \text{Atomic mode 1}; \mathsf{Update}(\mathsf{now})).(h = 0 \wedge y \leq b \wedge p) \\ & \vee \\ & \ldots \\ & \vee \\ & sp.(d := e; h := 1; \mathsf{Update}(\mathsf{now})).(sm = \text{Atomic mode 2} \wedge y = c \wedge p) \end{array}$$

Assuming that $\mathsf{sp}.\{\mathsf{y} \leq \mathsf{c}\}.\mathsf{p} \equiv y \leq c \wedge (\exists \mathsf{x} \cdot \mathsf{p}(\mathsf{x}))$, the above sp can be mechanically computed by successively applying rules (1) - (2). The correctness proofs reduce to checking whether each of the strongest postconditions of the above disjunction implies the requirement q, given earlier.

In service-oriented systems, there is often the case that services need to synchronize their behaviors. In order to model synchronized behavior, we introduce a special kind of REMES mode, given in Figure 2, which can act either as an AND mode, or as an OR mode, depending on whether the services need to be entered simultaneously, or not.

The composite mode of Figure 2 contains as sub-modes the services that need to be synchronized. For AND modes, both Service a, and Service b are entered at the same time (through their entry point). This means that the edges marked with (*) do not have guards. In case of OR modes, one or all constituent services are entered, so the edges marked with (*) are annotated with guards. If some of the edges need to be taken at the same time in both services, the communication between Service a and b is realized via synchronization variables, $\mathsf{chan}\ (\mathsf{in}\ \mathsf{x}), (\mathsf{out}\ \mathsf{x})$, which are used similarly to

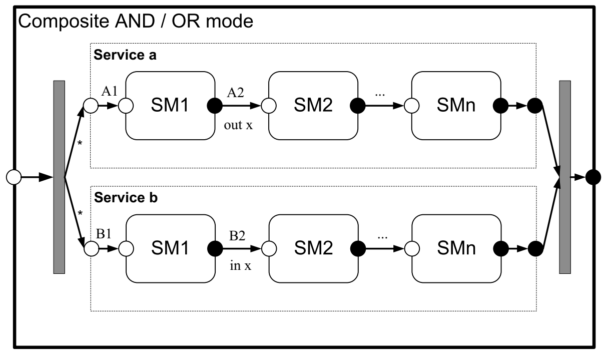

Fig. 2. AND/OR REMES mode.

the PTA channels $x?, x!$, respectively. Depending on the required synchronization type and starting time of the constituent services' execution, AND modes, but also OR can be employed when either "**and**" synchronization (both services should finish execution at the same time), or "**max**" synchronization (the composite mode finishes when the slowest service finishes) is required.

In Figure 2, Service a, Service b need to synchronize actions $A2$, $B2$. This can be done by decorating the respective edges with channel variables $out\ x$ for $A2$, and $in\ x$ for $B2$, meaning that the respective edges are taken simultaneously in both services, $A2$ writing variables that $B2$ is reading. The same applies if the services need to "and"-synchronize at the end of their execution. The exit edge of each service, respectively, needs to be annotated with chan variables.

The GCL representation of such synchronization requires strengthening the guards of the respective synchronized commands of the conditional statement, as follows: $(in\ x) \wedge g_{A2} \rightarrow S_{A2}$, $(out\ x) \wedge g_{B2} \rightarrow S_{B2}$, where S_{A2}, S_{B2} are the action bodies of $A2$, $B2$, respectively. The actions can then be executed in a sequence, with the one writing variables, first. The "**max**" synchronization can be represented in GCL by using a virtual selector (variable sel) [18], which selects for execution the modes $SM1, \ldots, SMn$, according to the control flow, marks them as executed after they finish their execution, and keeps the time values of now in a copy variable now_c, which is updated only after the slowest service finishes executing; the latter translates in exiting the composite AND, or OR mode.

4 Hierarchical Language for Dynamic Service Composition: Syntax and Semantics

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, whose behavior is 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:

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

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

– **Create service**: *create service_name*
$[pre]$: $service\_name = \mathsf{NULL}$
create : $Type \times N \times N \times {''passive''} \times (\Sigma \rightarrow bool) \times (\Sigma \rightarrow bool) \rightarrow service\_name$
$\{post\}$: $service\_name \neq \mathsf{NULL}$

– **Delete service**: *del service_name*
$[pre]$: $service\_name \neq \mathsf{NULL}$
del : $service\_name \rightarrow \mathsf{NULL}$
$\{post\}$: $service\_name = \mathsf{NULL}$

– **Create service list**: *create s_list*
$[pre]$: $s\_list = \mathsf{NULL}$
create_list : $s\_list \rightarrow s\_list,\ s\_list = List()$
$\{post\}$: $s\_list \neq \mathsf{NULL}$

– **Delete service list**: *del s_list*
$[pre]$: $s\_list \neq \mathsf{NULL}$
del_list : $s\_list \rightarrow \mathsf{NULL}$
$\{post\}$: $s\_list = \mathsf{NULL}$

– **Add service to a list**: *add service_name, s_list*
$[pre]$: $service\_name \notin s\_list$
add : $s\_list \rightarrow s\_list$
$\{post\}$: $service\_name \in s\_list$

– **Remove service from the list**: *del service_name, s_list*
$[pre]$: $service\_name \in s\_list$
del : $s\_list \rightarrow s\_list$
$\{post\}$: $service\_name \notin s\_list$

– **Replace service in the list**: $replace\ service\_name_1, service\_name_2$
$[pre]$: $s\_list(p) = service\_name_1$
replace : $s\_list \rightarrow s\_list$
$\{post\}$: $s\_list(p) = service\_name_2$

– **Insert service at a specific position**: $insert\ service\_name_i, s\_list$
$[pre]$: $s\_list(p) \neq service\_name_i$
add : $s\_list \rightarrow s\_list$
$\{post\}$: $s\_list(p) = service\_name_i$

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 ($s\_list = \mathsf{List : String([``Shuttle1'', ``Shuttle2''])}$). Also, adding a service to a list means, in this context, appending that service, that is, adding it at the end of the list. Replacing a service with another one, and inserting a service at a specific position requires the use of parameter p, which specifies the position at which the service is replaced or inserted.

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:

$$\mathsf{DCL} ::= (\mathsf{s\_list}, \mathsf{PROTOCOL}, \mathsf{REQ})$$

$$\mathsf{HDCL} ::= (((\mathsf{DCL}^{+}, \mathsf{PROTOCOL}, \mathsf{REQ})^{+}, \mathsf{PROTOCOL}, \mathsf{REQ})^{+}, \ \ldots\ )$$

The formula above allows a theoretically infinite degree of nesting. 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:

$$\mathsf{PROTOCOL} ::= \mathsf{unary\_operator}\ service\_name \mid service_m\ \mathsf{binary\_operator}\ service_n$$

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:

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

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:

$$status_{s_1} = active \ \wedge \ status_{s_2} \ = \ idle \ \wedge \ Post_{s_1} \Rightarrow (status_{s_2} \ = \ active)$$

If we assume n services $s_1, \ldots, s_n$ of a list, executing s_1 first is defined as:

$$\mathsf{Exec-first\ s_1} \triangleq \mathsf{s_1} \ \| \ \neg \mathsf{g_{s_1}} \rightarrow (\mathsf{s_2\ Binary\_operator \ldots Binary\_operator\ s_n})$$

This means that, even if any other service (or service composition) could be executed, it will be executed only after s_1 has finished execution.

– **Sequential composition** - two services are executed in a sequence, uninterrupted, e.g., $s_1; s_2$. The correctness condition of $s_1; s_2$ is:

$$(sp.s_2.(sp.s_1.Pre_{s_1}) \Rightarrow Post_{s_2}) \wedge (Post_{s_2} \Rightarrow REQ)$$

– **Parallel composition's** $(s_1 \parallel s_2)$ correctness condition is:

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

– **Parallel composition with synchronization** - we denote by S-AND the set of services belonging to an AND mode, which need to synchronize their executions in the end. Then, the "and" synchronization of such services is defined as:

$$(s_1 \parallel_{SYNC-and} s_2) \triangleq (s_1, s_2 \in \mathsf{S-AND} \Rightarrow ((\forall\, now \cdot status_{s_1} = status_{s_2} = active) \\ \wedge (start_{s_1} + TimetoServe_{s_1} = start_{s_2} + TimetoServe_{s_2})))$$

The correctness condition of the "and-AND" synchronization is given below:

$$(sp.(s_1 \parallel_{SYNC-and} s_2).Pre_{AND} \Rightarrow (Post_{s_1} \wedge Post_{s_2})) \wedge (Post_{s_1} \wedge Post_{s_2} \Rightarrow REQ)$$

A service user, but also a developer of services, might need to replace a service with one with possibly better QoS. 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. Either weakening the service precondition or strengthening its postcondition qualifies as service refinement.

5 Example: An Autonomous Shuttle System

In this section, we consider an example, previously modeled and analyzed in the PTA framework, in our recent work [5].

We consider a simplified version of a three train system that provides transportation service to three different locations. The system has been developed at University of Paderborn within the Railcab project [11]. While in our previous work [5], we have focused on resource effective design, in the current example, we extract parts of the behavior described by Giese and Klein [11], to show how services are created, invoked, composed, and idled, by using the REMES extended interface and behavioral language.

Each of the trains has a well-defined path to follow, as shown in Fig. 3. During the transport, the shuttles might meet at point B, in which they are forced to create a convoy.

In order to enter the convoy, they have to respect given speed and acceleration limits, measured in points A1, A2, and A3, respectively, 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 the sensor at point C, in Fig. 3, has sent the “safe to continue” signal.

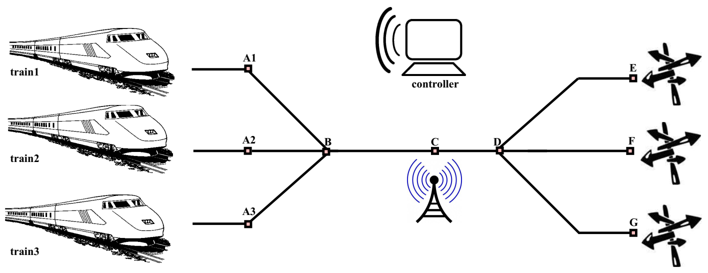

Fig. 3. An example overview.

After the destination point is being reached, a shuttle is free to turn to the idle state, and wait for new orders. The system described above is equipped with one central controller, as shown in Fig. 3, which decides when and which shuttle to invoke, based on the service descriptions for each shuttle, respectively.

5.1 Modeling the Shuttle System in REMES

We model the behavior of the Autonomous shuttle system services as modes in the extended REMES. The composite mode of Shuttle1 is depicted in Fig. 4, yet, due to lack of space, we do not show here the constituent submodes, but we briefly explain them instead (for more details we refer reader to [5]).

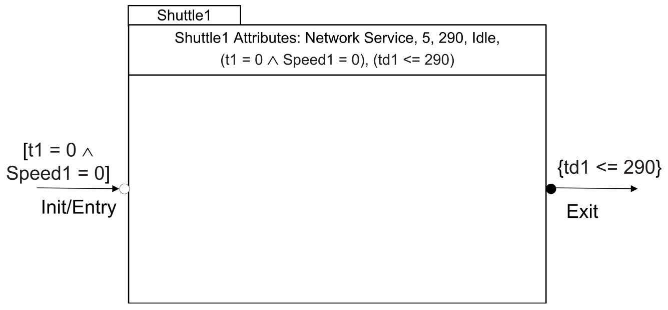
Shuttle1
Shuttle1 Attributes: Network Service, 5, 290, Idle,
(t1 = 0 ∧ Speed1 = 0), (td1 <= 290)
[t1 = 0 ∧
Speed1 = 0]
Init/Entry
{td1 <= 290}
Exit

Fig. 4. The model of Shuttle1 given as a REMES service.

The mode consists of the *atomic* modes (i.e., Acceleration1, STOP, and Destination). They communicate data between each other using the global variables: speed$_i$, status$_i$,

t_i, and StatusConvoy. The control interfaces are used to expose mode attributes relevant for mode discovery. Shuttle1 and Shuttle3 have the same behavior, while Shuttle2 is an older shuttle than the other two, and therefore it requires more time to start, accelerate, slow down.

5.2 Applying the Hierarchical Language

Below, we illustrate the use of our proposed hierarchical language for modeling service composition, as depicted in Table 1, on the example described in Section 5.

Table 1. An illustration of the REMES language

00 **declare** Shuttle1 ::=< network service,	18 **create** Shuttle1
01 5,	19 **create** Shuttle2
02 290,	20 **create** Shuttle3
03 passive,	21 **create** list_Convoy
04 $(t1 = 0 \wedge speed = 0)$,	22 **add** Shuttle1 list_Convoy
05 $(t1 \leq 290) >$	23 **add** Shuttle2 list_Convoy
06 **declare** Shuttle2 ::=< network service,	24 **DCL**_Convoy ::= (list_Convoy, ; , $t \leq 300$)
07 7,	25 **HDCL**_Convoy ::= ((DCL_Convoy, Shuttle3), $\Vert$, $t \leq 300$)
08 300,	26 **check**(sp.(Shuttle1; Shuttle2).$(t1 = 0 \wedge speed = 0) \wedge (t = t1 \vee t = t2)) \Rightarrow (t \leq 300)$
09 passive,	27 **check**(sp.Shuttle3.$(t3 = 0 \wedge speed = 0) \wedge (t = t3)) \Rightarrow (t \leq 300)$
10 $(t2 = 0 \wedge speed = 0)$,	28 **del** HDCL_Convoy
11 $(t2 \leq 300) >$	
12 **declare** Shuttle3 ::=< network service,	
13 5,	
14 290,	
15 passive,	
16 $(t3 = 0 \wedge speed = 0)$,	
17 $(t3 \leq 290) >$	

The needed services are introduced through the declarative part (lines 00-17 in Table 1). A service declaration contains the 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 Table 1), and added to the service list using the *add* command (lines 22-23 in Table 1). 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, through HDCL, as shown in line 25 of Table 1. 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, by forward analysis, e.g., by calculating the strongest postcondition of a given composition w.r.t. a given precondition. Due to space limitation, we show only the final computed result. Below, $p1 \equiv (t1 = 0 \wedge speed = 0)$.

By applying the sp rules (1) - (3), we get the following:

$$\begin{aligned} sp.(Shuttle1; Shuttle2).p1 &\equiv sp.Shuttle2.(sp.Shuttle1.p1) \\ sp.Shuttle1.p1 &\equiv (t1 = 0 \wedge 245 \leq t \leq 266 \wedge speed1 = 0 \wedge \\ &\quad \wedge mode = Destination \wedge r1 = 0 \wedge \\ &\quad \wedge status1 = end1 = idle) \end{aligned}$$

$$\begin{aligned}\mathsf{sp.Shuttle2.(sp.Shuttle1.p1)} \equiv\ & (\mathsf{t1} = \mathsf{t2} = 0 \wedge 264 \leq \mathsf{t} \leq 285 \wedge \\ & \wedge \mathsf{speed1} = \mathsf{speed2} = 0 \wedge \mathsf{r1} = \mathsf{r2} = 0 \wedge \\ & \wedge \mathsf{status1} = \mathsf{end1} = \mathsf{idle} \wedge \mathsf{status2} = \mathsf{end2} = \mathsf{idle})\end{aligned}$$

One can notice that the requirement $\mathsf{REQ} \equiv (\mathsf{t} \leq 300)$ is implied by the calculated strongest postcondition to which the condition $(\mathsf{t} = \mathsf{t1} \vee \mathsf{t} = \mathsf{t2})$ is added. This is actually what the command check should return as a main proof obligation, provided that the method is implemented in the REMES tool-chain.

For the second check, we have Shuttle3 composed in parallel with the sequential composition of the other two shuttles, with $\mathsf{p3} \equiv (\mathsf{t3} = 0 \wedge \mathsf{speed} = 0)$. Then, according to the composition semantics of section 4, proving the correctness of the $(\mathsf{Shuttle3}\ ||(\mathsf{Shuttle1}; \mathsf{Shuttle2}))$ composition reduces to showing that:

$$(\mathsf{sp.Shuttle3.p3} \vee \mathsf{sp.Shuttle2.(sp.Shuttle1.p1)}) \Rightarrow \mathsf{REQ}$$

As already shown, the sequential composition of the first two shuttles implies the requirement. What is left to be proven is that the strongest postcondition of Shuttle3, w.r.t p3, also implies the requirement. The calculated strongest postcondition of the latter is as follows:

$$\begin{aligned}\mathsf{sp.Shuttle3.p3} \equiv\ & (\mathsf{t3} = 0 \wedge 245 \leq \mathsf{t} \leq 266 \wedge \mathsf{speed3} = 0 \wedge \\ & \wedge \mathsf{mode} = \mathsf{Destination} \wedge \mathsf{r3} = 0 \wedge \mathsf{status3} = \mathsf{end3} = \mathsf{idle})\end{aligned}$$

It is easy to check that the requirement REQ is actually implied by $\mathsf{sp.Shuttle3.p} \wedge \mathsf{t} = \mathsf{t3}$. This concludes our service composition correctness verification.

6 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 [14] 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 does not provide means for detailed analysis, as the one supported by REMES.

The third group includes approaches with formal background. Rychlý describes the service behavior as a component-based system for dynamic architectures [16]. 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 can also cater for service descriptions including timing and resource annotations [5]. Foster et al. present an approach for modeling and analysis of web service compositions [10]. The approach takes BPEL4WS service specification and translates it into Finite State Processes (FSP), and Labeled Transition Systems (LTS), for analysis purposes.

A comprehensive survey on several approaches that are accommodating service composition, and are checking the correctness of compositions [3, 12, 14, 15] is given by Beek et al. [20]. Regarding service modeling, all these approaches are solid; however, w.r.t. service compositions and their correctness checking [7, 13, 17] (usually by employing formal methods), such approaches show limited capabilities to automatically support these processes. In comparison, as shown in this paper, compositions of REMES models can be mechanically reasoned about (although, as for now, we still miss the interface correctness tool support), or can be automatically translated to timed- [1] or priced timed automata [2], and analyzed with UPPAAL , or UPPAAL CORA tools [1], for functional but also extra-functional behaviors (timing and resource-wise behaviors).

7 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. Even if the original semantics of REMES [19] is given in terms of Priced Timed Automata (PTA), here, we have chosen to use Hoare triples and the strongest postcondition semantics to prove service correctness, motivated by the lack of decidability results for computing simulations relations on PTA. 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.

As future work, we plan to look into the algorithmic computation of strongest postconditions of priced timed automata, by building on preliminary results of Badban et al. [4]. We also intend to extend the REMES tool-chain with a postcondition calculator.

References

[1] Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994), citeseer.nj.nec.com/alur94theory.html

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

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

[4] Badban, B., Leue, S., Smaus, J.G.: Automated predicate abstraction for real-time models. EPTCS 10, 36 (2009), doi:10.4204/EPTCS.10.3

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

[5] Causevic, A., Seceleanu, C., Pettersson, P.: Formal reasoning of resource-aware services. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-245/2010-1-SE, Mälardalen University (June 2010)

[6] Causevic, A., Vulgarakis, A.: 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)

[7] Daz, G., Pardo, J.J., Cambronero, M.E., Valero, V., Cuartero, F.: Automatic translation of ws-cdl choreographies to timed automata. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) EPEW/WS-FM. Lecture Notes in Computer Science, vol. 3670, pp. 230–242. Springer (2005)

[8] Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)

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

[10] Foster, H., Emmerich, W., Kramer, J., Magee, J., Rosenblum, D., Uchitel, S.: 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. pp. 225–234. ACM, New York, NY, USA (2007)

[11] Giese, H., Klein, F.: Autonomous shuttle system case study. In: Scenarios: Models, Transformations and Tools. pp. 90–94 (2003)

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

[13] Narayanan, S., McIlraith, S.A.: Simulation, verification and automated composition of web services. In: WWW '02: Proceedings of the 11th international conference on World Wide Web. pp. 77–88. ACM, New York, NY, USA (2002)

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

[15] Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web service modeling ontology. Applied Ontology 1(1), 77–106 (2005)

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

[17] Salaün, G., Bordeaux, L., Schaerf, M.: Describing and reasoning on web services using process algebra. In: ICWS '04: Proceedings of the IEEE International Conference on Web Services. p. 43. IEEE Computer Society, Washington, DC, USA (2004)

[18] Seceleanu, C.: A Methodology for Constructing Correct Reactive Systems. Ph.D. thesis, Turku Centre for Computer Science (TUCS) (December 2005)

[19] Seceleanu, C., Vulgarakis, A., Pettersson, P.: 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)

[20] Ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Formal methods for service composition. Annals of Mathematics, Computing & Teleinformatics 1(5), 1 – 10 (2007), http://journals.teilar.gr/amct/, in: Annals of Mathematics, Computing & Teleinformatics, vol. 1 (5) pp. 1 - 10. Technological Education Institute of Larissa (TEIL), Greece, 2007.