A

Automated Synthesis of Local Time Requirement for Service Composition

Tian Huat Tan, Singapore University of Technology and Design, Singapore
Étienne André, Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France and École Centrale de Nantes, IRCCyN, CNRS, UMR 6597, France
Manman Chen, Singapore University of Technology and Design, Singapore
Jun Sun, Singapore University of Technology and Design, Singapore
Yang Liu, Nanyang Technological University, Singapore
Jin Song Dong, National University of Singapore, Singapore

Service composition aims at achieving a business goal by composing existing service-based applications or components. The response time of a service is crucial especially in time critical business environments, which is often stated as a clause in service level agreements between service providers and service users. To meet the guaranteed response time requirement of a composite service, it is important to select a feasible set of component services such that their response time will collectively satisfy the response time requirement of the composite service. In this work, we propose a fully automated approach to synthesize the response time requirement of component services, in the form of a constraint on the local response times. The synthesized requirement will guarantee the satisfaction of the global response time requirement, statically or dynamically. We implemented our work into a tool SELAMAT, and performed several experiments to evaluate the validity of our approach.

General Terms: Web Service Composition, Parameter Synthesis, Labeled Transition System

1. INTRODUCTION AND MOTIVATION

Service-oriented architecture is a paradigm where building blocks are used as services for software applications. Services encapsulate their functionalities, information, and make them available through a set of operations accessible over a network infrastructure using standards like SOAP [Gudgin et al. 2007] and WSDL [Chinnici et al. 2007]. To make use of a set of services to achieve a business goal, service composition languages such as BPEL (Business Process Execution Language) [Alves et al. 2007] have been proposed. A service that is composed by other services is called a *composite* service, and services that the composite service makes use of are called *component* services.

The requirement on the service response time is often an important clause in service-level agreements (SLAs) especially in business where timing is critical. An SLA is a contract between service consumers and service providers specifying the expected quality of service (QoS) level. Henceforth, we denote the response time requirement of composite services as *global time requirement*, and the set of constraints on the response times of the component services as *local time requirement*. The response time of a composite service is highly dependent on that of each component service. It is therefore crucial to derive local time requirements (*i. e.*, requirements for the component services) from the global time requirement so that it will help in the selection of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© YYYY ACM. 1539-9087/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

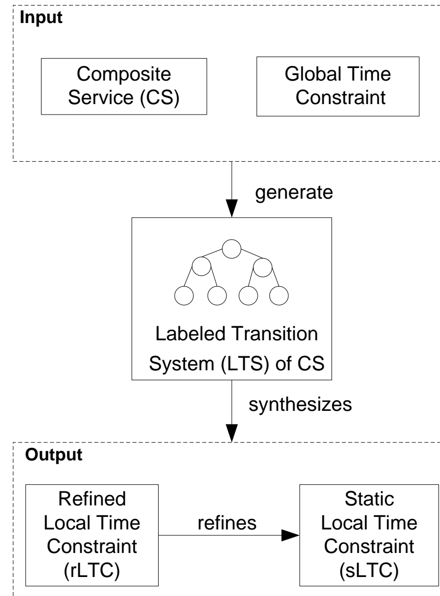


Fig. 1: General approach

component services when building a composite service while satisfying the response time requirement.

Consider an example of a stock indices service, which has an SLA with the subscribed users requiring that the stock indices shall be returned within three seconds upon request. The stock indices service makes use of several component services, including a paid service, for requesting stock indices. The stock indices service provider would be interested in knowing the local time requirement of the component services, while satisfying the global response time requirement. To avoid discarding any service candidates that might be part of a feasible composition, the synthesized local time requirement needs to be as *weak* as possible, *i. e.*, to contain as many values for local time requirements as possible. This is crucial as having a faster service might incur a higher cost.

1.1. Contribution

In this paper, we present a fully automated technique to synthesize the local time requirement in composite services. Our approach performs an analysis of the composite service's behavior, using techniques inspired by parameter synthesis for real-time systems. Our synthesis approach does not only avoid bad scenarios in the service composition, but also guarantees the fulfillment of the global time requirement.

We use as a formalism BPEL, which is a *de-facto* standard for service composition. BPEL supports control flow structures that involve complex timing constructs (*e. g.*, `<pick>` control structure) and concurrent execution of activities (*e. g.*, `<flow>` control structure). Due to the non-determinism in both time and control flow, it is unknown which execution path will be executed at runtime. Such a combination of timing constructs, concurrent calls to external services, and complex control structures, makes it a challenge to synthesize the local time requirement.

Fig. 1 illustrates the main steps of our approach for synthesizing local time requirements, described in the following. The required inputs are the specification of the composite service *CS*, and its global time requirement. The output will be local time requirements (at design time, and then at runtime) given in the form of a linear constraint.

First contribution. We first propose a semantics for BPEL composite services augmented with timing parameters, *i. e.*, constants the value of which is not known at design time; this symbolic semantics is given the form of a labeled transition system (LTS).

Second contribution. Based on the LTS resulting from the input composite service, we propose an approach to synthesize local time requirements of component services, represented as a (linear) constraint, which we refer to as the *local time constraint*. During the design phase of a composite service, the local time constraint is synthesized based on *all* possible execution paths, since it is unknown which execution path will be executed at runtime (this will depend in the dynamic behavior of the system, and in particular the evaluation of guards). The local time constraint of a composite service that is synthesized during the design time is called the *static local time constraint* (hereafter sLTC).

The synthesized sLTC has several advantages. Firstly, it allows the selection of feasible services when creating a new composite service, from a large pool of services with similar functionalities but different local response times. Secondly, service designers can use the synthesized result to avoid over-approximations on the local response times, which may lead the service provider to purchase a service at a higher cost, while a service at a lower cost with a slower response time may be sufficient to guarantee the global time requirement. Thirdly, the local time requirements serve as a safe guideline when component services need to be substituted or new services need to be introduced.

Third contribution. Due to the highly evolving and dynamic environment the composite service is running in, the design time assumptions for Web service composition, even if they are initially accurate, may later change at runtime. For example, the execution time of a component service could violate the sLTC due to reasons such as network congestion. Nevertheless, this does not necessarily imply that the composite service will not satisfy the global time requirement. Indeed, the sLTC is synthesized based on all possible execution paths at design time, whereas only one path will be executed at runtime. At runtime, some of the execution paths can be eliminated. Therefore, we can use the runtime information to refine the sLTC to make it weaker – which results in a more relaxed constraint on the response times of the component services. We refer to the sLTC refined at runtime as a *refined local time constraint* (hereafter rLTC). The rLTC is then used to decide whether the current composite service can still satisfy the global time requirement.

Fourth contribution. We implemented our algorithms into a tool SELAMAT. We then conducted experiments on several case studies, that show that the rLTC can indeed help to improve the sLTC. In addition, we show that the runtime adaptation does not incur much overhead in practice.

About this manuscript. This manuscript is a substantially improved version of [Tan et al. 2013]. In [Tan et al. 2013], we presented a method supporting the synthesis of sLTC from the global time requirement. We provided an extra analysis on the BPEL activities, and classified them as “or-activities” or “and-activities”. We then extended labeled transition systems (LTS) with and-states and or-states, which we called *and/or*

LTS (AOLTS), for synthesizing the local time requirement. The main contribution of this manuscript is the introduction of rLTC to verify whether the service composition could satisfy the global time requirement at runtime. Additionally, this manuscript also improves the approach for synthesizing sLTC presented in [Tan et al. 2013], by removing the and/or states in the LTS. This results in a more comprehensible semantics model and more straightforward synthesis algorithms for local time requirement. We also implemented our work in SELAMAT and conducted extensive evaluations, that were not present in [Tan et al. 2013].

1.2. Outline

The rest of this paper is structured as follows. Section 2 provides the necessary definitions and terminologies. Section 3 introduces a timed BPEL running example. Section 4 introduces our approach to analyze the BPEL process. Section 5 presents the synthesis algorithms for sLTC. Section 6 introduces rLTC, and its usage for runtime adaptation of a service composition. Section 7 presents our implementation and evaluates our approach using four case studies. Section 8 reviews related works. Finally, Section 9 concludes the paper, and outlines future work.

2. A FORMAL MODEL FOR PARAMETRIC COMPOSITE SERVICES

In this section, we introduce the concepts used throughout the paper.

2.1. Variables, Clocks, Parameters, and Constraints

We assume *AllVars* to be the universal set of finite-domain *variables*. Given a finite set $\mathcal{V} \subset \mathit{AllVars}$, a *variable valuation* for $\mathcal{V}$ is a function assigning to each variable a value in its domain. We denote by $\mathit{Valuations}(\mathcal{V})$ the set of all variable valuations of $\mathcal{V}$. Given a variable $y \in \mathcal{V}$ and a variable valuation $v \in \mathit{Valuations}(\mathcal{V})$, we denote by $v(y) = \perp$ the fact that variable y is uninitialized in valuation v.

The clocks, parameters and constraints that we use in this work are similar to the ones used in the formalisms of timed automata [Alur and Dill 1994], parametric timed automata [Alur et al. 1993] and stateful timed CSP [Sun et al. 2013]. A clock is a variable with values in the set of non-negative real numbers $\mathbb{R}_{\geq 0}$; a clock is used here to record the time passing of activities. All clocks are progressing at the same rate (just as in timed automata [Alur and Dill 1994]). Let *AllClocks* denote the universal set of clocks, disjoint with *AllVars*. Let $X = \{x_1, \dots, x_H\} \subset \mathit{AllClocks}$ (for some integer H) be a finite set of clocks. A *clock valuation* is a function $w : X \to \mathbb{R}_{\geq 0}$, that assigns a non-negative real value to each clock.

A *parameter* is an unknown constant, used here to represent the unknown response time of a component service. Let *AllParams* denote the universal set of parameters, disjoint with *AllClocks* and *AllVars*. Given a finite set of parameters $U = \{u_1, \dots, u_m\} \subset \mathit{AllParams}$ (for some integer m), a *parameter valuation* is a function $\pi : U \to \mathbb{Q}_{\geq 0}$ assigning a non-negative rational value to each parameter.

A *linear term* over $X \cup U$ is an expression of the form $\sum_{1 \leq i \leq N} \alpha_i z_i + d$ for some $N \in \mathbb{N}$, with $z_i \in X \cup U$, $\alpha_i \in \mathbb{Q}_{\geq 0}$ for $1 \leq i \leq N$, and $d \in \mathbb{Q}_{\geq 0}$. We denote by $\mathcal{L}_{X \cup U}$ the set of all linear terms over X and U. Similarly, we denote by $\mathcal{L}_X$ (resp. $\mathcal{L}_U$) the set of all linear terms over X (resp. U). An *inequality* over X and U is $e \prec e'$ with $\prec \in \{<, \leq\}$, where e, $e' \in \mathcal{L}_{X \cup U}$.

A *convex constraint* (or *constraint*) is a conjunction of inequalities. We denote by $\mathcal{C}_{X \cup U}$ the set of all convex constraints over X and U. Similarly, we denote by $\mathcal{C}_X$ (resp. $\mathcal{C}_U$) the set of all convex constraints over X (resp. U). A *non-necessarily convex constraint* (or

NNCC) is a conjunction of disjunction of inequalities[1]; NNCCs are used in this paper to represent the synthesized local time constraint obtained via the methods proposed in this paper. Note that the negation of an inequality remains an inequality; however, the negation of a convex constraint becomes (in the general case) an NNCC. We denote by $\mathcal{NC}_U$ the set of all NNCCs over U.

Henceforth, we use w (resp. π) to denote a clock (resp. parameter) valuation. Let $C \in \mathcal{NC}_U$, $C[\pi]$ denotes the valuation of C with π, *i. e.*, the constraint over X obtained by replacing each $u \in U$ with $\pi(u)$ in C. Note that $C[\pi]$ can be written as $C \wedge \bigwedge_{u_i \in U} u_i = \pi_i$.

We say that π *satisfies* C, denoted by $\pi \models C$, if $C[\pi]$ evaluates to true. C is *empty* if there does not exist a parameter valuation π such that $\pi \models C$; otherwise C is *non-empty*. We define $C^{\uparrow}$ as the *time elapsing* of C, *i. e.*, the constraint over X and U obtained from C by delaying all clocks of an arbitrary amount of time d. Given two constraints $C_1, C_2 \in \mathcal{NC}_U$, we say that C_2 is *weaker* (or *more relaxed*) than C_1, denoted by $C_1 \subseteq C_2$, if $\forall \pi : \pi \models C_1 \Rightarrow \pi \models C_2$. Similarly, C_2 is *strictly weaker* (or *strictly more relaxed*) than C_1, denoted by $C_1 \subset C_2$, if $C_1 \subseteq C_2$ and $C_1 \neq C_2$.

Given $C \in \mathcal{C}_{X \cup U}$ and $X' \subseteq X$, we denote by $prune_{X'}(C)$ the constraint in $\mathcal{C}_{X \cup U}$ that is obtained from C by pruning the clocks in X'; this can be achieved using variable elimination techniques such as Fourier-Motzkin (see, *e. g.*, [Schrijver 1986]). More generally, Given $C \in \mathcal{C}_{X \cup U}$, we denote by $C{\downarrow_U}$ the *projection* of constraint C onto U, *i. e.*, the constraint obtained from C by pruning all clock variables. Again, such projections can be computed using Fourier-Motzkin elimination.

2.2. Syntax of Composite Service Processes

BPEL [Alves et al. 2007] is an industrial standard for implementing composition of existing Web services by specifying an executable workflow using predefined activities. In this work, we assume the composite service is specified using the BPEL language. Basic BPEL activities that communicate with component Web services are `<receive>`, `<invoke>`, and `<reply>`, which are used to receive messages, invoke an operation of component Web services and return values respectively. We denote them as *communication activities*. The control flow of the service is defined using structural activities such as `<flow>`, `<sequence>`, `<pick>`, `<if>`, etc.

A composite service CS makes use of a finite number of component services to accomplish a task. Let $E = \{S_1, \ldots, S_n\}$ be the set of all component services that are used by CS. In this work, we assume that the response time of a composite service is based on the time spent on individual communication activities, and the time incurred by internal operations of the composite service is negligible.[2]

Composite services are expressed using *processes*. We define a formal syntax definition in Fig. 2, where S is a component service, P and Q are composite service processes, b is a Boolean expression, and $a_i \in \mathbb{Q}_{>0}$ are positive rational numbers, for $1 \leq i \leq k$.

Let us describe the BPEL syntax notations below:

— *rec*(S) and *reply*(S) are used to denote "receive from" and "reply to" a service S, respectively;
— *sInv*(S) (resp. *aInv*(S)) denotes the synchronous (resp. asynchronous) invocation of a component service S;
— $P \mid\mid\mid Q$ denotes the concurrent composition of BPEL activities P and Q;
— $P\,\mathbf{;}\,Q$ denotes the sequential composition of BPEL activities P and Q;

[1]Without loss of generality, we assume here that all NNCCs are in conjunctive normal form (CNF).
[2]We discuss the time incurred for internal operations in Section 6.6.

$P \mathrel{\widehat{=}} \mathit{rec}(S)$	receive activity
$\mid \mathit{reply}(S)$	reply activity
$\mid \mathit{sInv}(S)$	synchronous invocation
$\mid \mathit{aInv}(S)$	asynchronous invocation
$\mid P \mathbin{\vert\vert\vert} Q$	concurrent activity
$\mid P \mathbin{;} Q$	sequential activity
$\mid P \lhd b \rhd Q$	conditional activity
$\mid \mathit{pick}(\biguplus_{i=1}^{n} S_i \Rightarrow P_i, \biguplus_{i=1}^{k} \mathit{alrm}(a_i) \Rightarrow Q_i)$	pick activity

Fig. 2: Syntax of composite service processes

— $P \lhd b \rhd Q$ denotes the conditional composition, where b is a guard condition on the process variables. If b evaluates to true, BPEL activity P is executed, otherwise activity Q is executed;

— $\mathit{pick}(\biguplus_{i=1}^{n} S_i \Rightarrow P_i, \biguplus_{i=1}^{k} \mathit{alrm}(a_i) \Rightarrow Q_i)$ denotes the BPEL *pick* composition, which contains two types of activities: *onMessage* activity and *onAlarm* activity. An *onMessage* activity $S_i \Rightarrow P_i$ is activated when the message from service S_i arrives and BPEL activity P_i is subsequently executed; an *onAlarm* activity $\mathit{alrm}(a_i) \Rightarrow Q_i$ is activated at a_i seconds, and BPEL activity Q_i is subsequently executed. The *pick* activity contains n *onMessage* activities and k *onAlarm* activities. Exactly one activity from these $n + k$ activities will be executed. If multiple activities activate at the same time, one of the activities will be chosen non-deterministically for execution. Given a *pick* activity P, we use $P.\mathit{onMessage}$ and $P.\mathit{onAlarm}$ to denote the *onMessage* and *onAlarm* branches of P respectively.

A *structural activity* is an activity that contains other activities. Concurrent, sequential, conditional, and pick activities are examples of structural activities. An activity that does not contain other activities is called an *atomic activity*, which includes receive, reply, synchronous invocation and asynchronous invocation activities. We denote by $\mathcal{P}_{np}$ the set of all possible (non-parametric) composite service processes.

Note that the communication activities can implicitly make use of variables for passing information. For example, let S be a component service that calculates the stock indices for a particular date. For synchronous invocation $\mathit{sInv}(S)$, it requires an input variable v_i that specifies the date information, and an output variable v_o to hold the return from $\mathit{sInv}(S)$. To keep the notations concise, we abstract the usage and assignment of variables for communication activities.

We introduce below an important assumption of this work:

ASSUMPTION 1. *All loops have a bound on the number of iterations and on the execution time.*

This assumption is necessary to ensure termination of our approach. We believe it is reasonable in practice (see Section 6.6 for a discussion).

2.3. Parametric Composite Service Models

Let us now formally define composite service models and parametric composite service models.

Definition 2.1 (*Composite Service Model*). A *composite service model CS* is a tuple $(\mathcal{V}, v_0, N_0)$, where $\mathcal{V} \subset \mathit{AllVars}$ is a finite set of variables, $v_0 \in \mathit{Valuations}(\mathcal{V})$ is an initial valuation that maps each variable to its initial value, and $N_0 \in \mathcal{P}_{np}$ is a composite

service process (defined according to the grammar of Fig. 2) making use of the variables in $\mathcal{V}$.

We now extend the definitions of services, composite service processes and composite service model to the parametric case. First, a parametric service is a service i, the response time of which is now a parameter $u_i \in U$, instead of a rational-valued constant. Then, a parametric composite service process is a service process defined according to the grammar of Fig. 2, where services ("*S*" in Fig. 2) are now parametric services. We denote by $\mathcal{P}$ the set of all possible parametric composite service processes. Finally, parametric composite service models are defined similarly to composite service models, except that the composite service processes are now parametric composite service processes.

Definition 2.2 (*Parametric Composite Service Model*). A *parametric composite service model* CS is a tuple $(\mathcal{V}, v_0, U, P_0, C_0)$, where $\mathcal{V} \subset \mathit{AllVars}$ is a finite set of variables; $v_0 \in \mathit{Valuations}(\mathcal{V})$ is an initial valuation that maps each variable to its initial value; $U \subset \mathit{AllParams}$ is a finite set of parameters; $P_0 \in \mathcal{P}$ is a parametric composite service process making use of the variables in $\mathcal{V}$ and $C_0 \in \mathcal{C}_U$ is the initial (convex) parametric constraint.

Process and model valuation. Given a parametric composite service process P with a parameter set $U = \{u_1, \dots, u_m\}$ and given a parameter valuation $(\pi(u_1), \dots, \pi(u_m))$, $P[\pi]$ denotes the *valuation* of P with π, *i.e.*, the process where each occurrence of a parameter u_i has been replaced with its valuation $\pi(u_i)$.

Given a parametric composite service model CS with a parameter set $U = \{u_1, \dots, u_m\}$, and given a parameter valuation $(\pi(u_1), \dots, \pi(u_m))$, $CS[\pi]$ denotes the *valuation* of CS with π, *i.e.*, the model $(\mathcal{V}, v_0, U, P_0, C)$, where C is $C_0 \wedge \bigwedge_{i=1}^{m}(u_i = \pi(u_i))$. Note that $CS[\pi]$ can be seen as a non-parametric service model $(\mathcal{V}, v_0, P_0[\pi])$.

2.4. Bad Activities

Given a BPEL service CS, we define a *bad activity* as an atomic activity such that its execution means that the composite service CS has violated the global time requirement. To distinguish bad activities, we allow the user to annotate a BPEL activity A as a bad activity. The annotation can be achieved, for example, by using extension attribute of BPEL activities.

3. A BPEL EXAMPLE WITH TIMED REQUIREMENTS

Let us introduce a *Stock Market Indices Service* (SMIS) that will be used as a running example. SMIS is a paid service and its goal is to provide updated stock indices to the subscribed users. It provides a service level agreement (SLA) to the subscribed users stating that it always responds within three seconds upon request.

SMIS has three component Web services: a database service (*DS*), a free news feed service (*FS*) and a paid news feed service (*PS*). The strategy of the SMIS is calling the free service *FS* before calling the paid service *PS* in order to minimize the cost. Upon returning the result to the user, the SMIS also stores the latest results in an external database service provided by *DS* (storage of the results is omitted here). The workflow of the SMIS is sketched in Fig. 3 in the form of a tree. When a request is received from a subscribed customer (`Receive User`), it synchronously invokes (*i.e.*, invoke and wait for reply) the database service (`Sync. Invoke DS`) to request stock indices stored in the past minute. Upon receiving the response from *DS*, the process is followed by an `<if>` branch (denoted by $\diamondsuit$). If the indices are available (`Indices exist`), then they are returned to the user (`Reply indices`). Otherwise, *FS* is invoked asynchronously (*i.e.*, the system moves on after the invocation without waiting for the reply). A `<pick>`

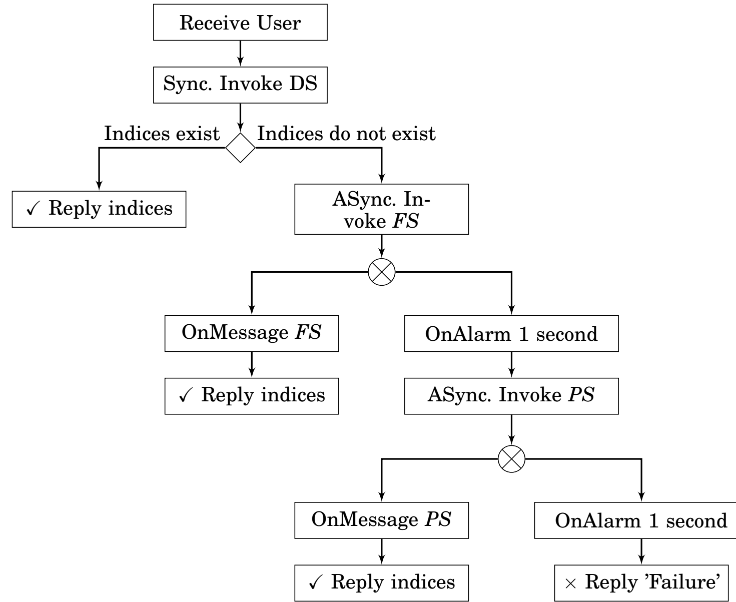


Fig. 3: Stock Market Indices Service

construct (denoted by $\bigotimes$) is used here to await an incoming response (`<onMessage>`) from previous asynchronous invocation and timeout (`<onAlarm>`) if necessary. If the response from *FS* (`OnMessage` *FS*) is received within one second, then the result is returned to the user (`Reply indices`). Otherwise, the timeout occurs (`OnAlarm 1 second`), and SMIS stops waiting for the result from *FS* and calls *PS* instead (`ASync. Invoke` *PS*). Similarly to *FS*, the result from *PS* is returned to the user, if the response from *PS* is received within one second. Otherwise, it notifies the user regarding the failure of getting stock indices (`Reply` '`Failure`'). The states marked with a ✓ (resp. ×) represent desired (resp. undesired) end states.

The global time requirement for SMIS is that SMIS should respond within three seconds upon request. It is of particular interest to know the local time requirements for services *PS*, *FS*, and *DS*, so as to fulfill the global time requirement. This information could also help to choose a paid service *PS* which is both cheap and responds quickly enough.

In this example, the bad activity is the reply activity that is triggered once after the component service *PS* fails to respond within one second.

4. A FORMAL SEMANTICS FOR PARAMETRIC COMPOSITE SERVICES

In this section, we provide our parametric composite service model with a formal semantics, given in the form of a labeled transition system (LTS). The semantics we use is inspired by the one proposed for (parametric) stateful timed Communicating Sequential Processes (CSP) [Sun et al. 2013; André et al. 2014], that makes use of implicit clocks.

We first recall LTSs (Section 4.1) and define symbolic states (Section 4.2). Following that, we define implicit clocks and the associated functions, *i. e.*, activation and idling

(Section 4.3). We then introduce our formal semantics (Section 4.4), and apply it to an example (Section 4.6).

4.1. Labeled Transition Systems

Definition 4.1 (*Labeled Transition System*). A *labeled transition system (LTS)* is a tuple $LTS = (Q, s_0, \Sigma, \delta)$, where

— Q is a set of states;
— $s_0 \in Q$ is the initial state;
— Σ is a set of actions;
— $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation.

Given $LTS = (Q, s_0, \Sigma, \delta)$, a state $s \in Q$ is a *terminal state* if there does not exist a state $s' \in Q$ and an action $a \in \Sigma$ such that $(s, a, s') \in \delta$; otherwise, s is said to be a *non-terminal state*. There is a *run* from a state s to state s', where s, $s' \in Q$, if there exists an alternating sequence of states and actions $\langle s_1, a_1, s_2, \ldots, a_{n-1}, s_n \rangle$, where $s_i \in Q$ for $1 \leq i \leq n$, $a_i \in \Sigma$ for $1 \leq i \leq n-1$, $s_1 = s$, $s_n = s'$, and $\forall i \in \{1, \ldots, n-1\}, (s_i, a_i, s_{i+1}) \in \delta$. A *complete run* is a run that starts in the initial state s_0 and ends in a terminal state. Given a state $s \in Q$, we use $succ(s)$ to denote the set of states reachable in one step from s; formally, $succ(s) = \{s' \mid \exists a \in \Sigma : s' \in Q \wedge (s, a, s') \in \delta\}$. Then, $succ^*(s)$ denotes the set of states reachable in one or more steps from s; formally, $succ^*(s) = \{s' \mid$ there is a run from s to $s'\}$.

Below, we introduce the notion of LTS starting from a state s as the LTS containing s and all its successor states and transitions.

Definition 4.2 (*sub-LTS*). Let $LTS = (Q, s_0, \Sigma, \delta)$ be an LTS, and let s be a state of Q. The *sub-LTS* of *LTS* starting from s is $(Q', s, \Sigma', \delta')$, where *i*) $Q' \subseteq Q$ is the set of states reachable from $s \in Q$ in *LTS* (*i.e.*, $succ^*(s)$ in *LTS*); *ii*) $\delta' \subseteq \delta$ is the transition relation satisfying the following condition: $s_1 \overset{a}{\hookrightarrow} s_2 \in \delta'$ if $s_1, s_2 \in Q'$ and $s_1 \overset{a}{\hookrightarrow} s_2 \in \delta$; and *iii*) $\Sigma' \subseteq \Sigma$ is the set of all actions used in δ', *i.e.*, $\{a \mid \exists s_1, s_2 \in Q' : s_1 \overset{a}{\hookrightarrow} s_2 \in \delta'\}$

4.2. Symbolic States

Let us define the notion of (symbolic) state of a parametric composite service model.

Definition 4.3. [State] Given a parametric composite service model $CS = (\mathcal{V}, v_0, U, P_0, C_0)$, a (symbolic) *state* of *CS* is a tuple $s = (v, P, C, D)$, where $v \in Valuations(\mathcal{V})$ is a valuation of the variables, P is a composite service process, C is a constraint over $\mathcal{C}_{X \cup U}$, and $D \in \mathcal{L}_U$ is the (parametric) elapsed time from the initial state s_0 to state s, excluding the idling time in state s.

Given a state $s = (v, P, C, D)$, we use the notation $s.v$ to denote the component v of s, and similarly for $s.P$, $s.C$ and $s.D$. When a parametric composite service model *CS* has no variable, we denote each state $s \in Q$ as (P, C, D) for the sake of brevity. Two states $s = (v, P, C, D)$ and $s' = (v', P', C', D')$ are *equal*, if $v = v'$, $P = P'$, $C = C'$, and $D = D'$.

4.3. Implicit Clocks

In order to provide parametric composite service models with a with real-time semantics, we use *clocks* to record the elapsing of time. Clocks are used to record the time elapsing in several formalisms, in particular in timed automata (TAs) [Alur and Dill 1994]. In TAs, the clocks are defined as part of the models and state space. It is known that the state space of the system may grow exponentially with the number of clocks and that the fewer clocks, the more efficient real-time model checking is [Bengtsson and Yi 2003]. In TAs, it is then possible to dynamically reduce the num-

$$
\begin{array}{llll}
Act(A(S), x) & = & A(S)_x & \text{A1} \\
Act(mpick, x) & = & mpick_x & \text{A2} \\
Act(A(S)_{x'}, x) & = & A(S)_{x'} & \text{A3} \\
Act(mpick_{x'}, x) & = & mpick_{x'} & \text{A4} \\
Act(P \oplus Q, x) & = & Act(P, x) \oplus Act(Q, x) & \text{A5} \\
Act(P\,;Q, x) & = & Act(P, x)\,;Q & \text{A6}
\end{array}
$$

where $A \in \{rec, sInv, aInv, reply\}$, $\oplus \in \{|||, \lhd b \rhd\}$,
$mpick = pick(\biguplus_{i=1}^{n} S_i \Rightarrow P_i, \biguplus_{i=1}^{k} alrm(a_i) \Rightarrow Q_i)$

Fig. 4: Activation function

ber of clocks [Daws and Yovine 1996]; the same can be done in parametric timed automata [André 2013]. An alternative approach is to define a semantics that creates clocks on the fly when necessary, and prunes them when no longer needed. This approach was initially proposed for stateful timed CSP [Sun et al. 2013]. This allows a smaller state space compared to the explicit clock approach; we refer to this second approach as the *implicit clock approach*. In this work, we use the implicit clock approach.

4.3.1. Clock Activation. Clocks are implicitly associated with timed processes. For instance, given a communication activity $sInv(S)$, a clock starts ticking once the activity becomes activated. To introduce clocks on the fly, we define an activation function *Act* in Fig. 4, in the spirit of the one defined in [Sun et al. 2013; André et al. 2014]. Given a process P, we denote by P_x the corresponding process that has been associated with clock x. When a new state s is reached, the activation function is called to assign a new clock for each newly activated communication activity. Rules A1 and A2 state that a new clock is associated with a BPEL communication activity A if A is newly activated. Rules A3 and A4 state that if a BPEL communication activity has already been assigned a clock, it will not be reassigned one. Rules A5 and A6 state that function *Act* is applied recursively to activated child activities for BPEL structural activities. For rule A6, function *Act* is applied only to activity P, but not to activity Q, since only activity P is executed next (activity Q will be executed only after the completion of activity P).

Given a process P, we denote by $aclk(P)$ the set of *active clocks* associated with P. For instance, the set of active clocks associated with process $P = sInv(S)_x \,|||\, sInv(S_1)_{x'}$ is $\{x, x'\}$.

4.3.2. Idling Function. In Fig. 5, we define the function *idle* that, given a state s, returns a constraint that specifies how long an activity can idle at state s. The result is a constraint over $X \cup U$. Rule I1 considers the situation when the communication requires waiting for the response of a component service S, and the value of clock x must not be larger than the response time parameter t_S of the service. Rule I2 considers the situation when no waiting is required. Rules I3 to I5 state that the function *idle* is applied recursively to activated child activities of a BPEL structural activity. Similar to rule A6, for rule I4, function *Act* is applied only to activity P, but not to activity Q, since only activity P is executed next. Therefore, given a state s and activity $P\,;Q$, we only need to consider how long the activity P can idle at state s.

4.4. Operational Semantics

We can now define the semantics of a parametric composite service model in the form of an LTS. Let $Y = \langle x_0, x_1, \cdots \rangle$ be a sequence of clocks.

$$\begin{array}{lcll} idle(A(S)_x) & = & x \leq t_S & \text{I1} \\ idle(B(S)_x) & = & (x = 0) & \text{I2} \\ idle(P \oplus Q) & = & idle(P) \wedge idle(Q) & \text{I3} \\ idle(P\,;Q) & = & idle(P) & \text{I4} \\ idle(mpick_x) & = & x \leq t_S \wedge x \leq a & \text{I5} \end{array}$$

where $A \in \{rec, sInv\}$, $B \in \{aInv, reply\}$, $\oplus \in \{|||, \lhd b \rhd\}$, $mpick = pick(\biguplus_{i=1}^{n} S_i \Rightarrow P_i, \biguplus_{i=1}^{k} alrm(a_i) \Rightarrow Q_i)$, and t_S is the parametric response time of service $mpick_x$.

Fig. 5: Idling function

$$\frac{}{(v, sInv(S)_x, C, D) \overset{e}{\hookrightarrow} (v', Stop, (x = t_S) \wedge C^{\uparrow}, D + t_S)} [\,rSInv\,]$$

$$\frac{}{(v, rec(S)_x, C, D) \overset{e}{\hookrightarrow} (v', Stop, (x = t_S) \wedge C^{\uparrow}, D + t_S)} [\,rRec\,]$$

$$\frac{}{(v, reply(S)_x, C, D) \overset{e}{\hookrightarrow} (v', Stop, (x = 0) \wedge C^{\uparrow}, D)} [\,rReply\,]$$

$$\frac{}{(v, aInv(S)_x, C, D) \overset{e}{\hookrightarrow} (v', Stop, (x = 0) \wedge C^{\uparrow}, D)} [\,rAInv\,]$$

let $mpick = pick(\biguplus_{i=1}^{n} S_i \Rightarrow P_i, \biguplus_{i=1}^{k} alrm(a_i) \Rightarrow Q_i)$

$$\frac{}{(v, mpick_x, C, D) \overset{e}{\hookrightarrow} (v', P_i, (x = t_i) \wedge idle(mpick_x) \wedge C^{\uparrow}, D + t_i)} [\,rPick1\,]$$

$$\frac{}{(v, mpick_x, C, D) \overset{e}{\hookrightarrow} (v', Q_i, (x = a) \wedge idle(mpick_x) \wedge C^{\uparrow}, D + a_i)} [\,rPick2\,]$$

$$\frac{v(b) = \bot}{(v, A \lhd b \rhd B, C, D) \overset{e}{\hookrightarrow} (v', A, C, D)} [\,rCond1\,]$$

$$\frac{v(b) = \bot}{(v, A \lhd b \rhd B, C, D) \overset{e}{\hookrightarrow} (v', B, C, D)} [\,rCond2\,]$$

$$\frac{v(b) = true}{(v, A \lhd b \rhd B, C, D) \overset{e}{\hookrightarrow} (v', A, C, D)} [\,rCond3\,]$$

$$\frac{v(b) = false}{(v, A \lhd b \rhd B, C, D) \overset{e}{\hookrightarrow} (v', B, C, D)} [\,rCond4\,]$$

$$\frac{(v, A, C, D) \overset{e}{\hookrightarrow} (v', A', C', D'), A' \neq Stop}{(v, A\,; B, C, D) \overset{e}{\hookrightarrow} (v', A'\,; B, C', D')} [\,rSeq1\,]$$

$$\frac{(v, A, C, D) \overset{e}{\hookrightarrow} (v', Stop, C', D')}{(v, A\,; B, C, D) \overset{\tau}{\hookrightarrow} (v', B, C', D')} [\,rSeq2\,]$$

$$\frac{(v, A, C, D) \overset{e}{\hookrightarrow} (v', A', C', D')}{(v, A \mid\mid\mid B, C, D) \overset{e}{\hookrightarrow} (v', A' \mid\mid\mid B, C' \wedge idle(B), D')} [\,rFlow1\,]$$

$$\frac{(v, B, C, D) \overset{e}{\hookrightarrow} (v', B', C', D')}{(v, A \mid\mid\mid B, C, D) \overset{e}{\hookrightarrow} (v', A \mid\mid\mid B', C' \wedge idle(A), D')} [\,rFlow2\,]$$

Fig. 6: Set of rules for the transition relation $\hookrightarrow$

Definition 4.4. Let $CS = (\mathcal{V}, v_0, U, P_0, C_0)$ be a parametric composite service model. The *semantics of CS* (hereafter denoted by LTS_{CS}) is the LTS (Q, s_0, Σ, δ) where

$$\begin{array}{l} Q = \{(v, P, C, D) \in Valuations(\mathcal{V}) \times \mathcal{P} \times \mathcal{C}_{X \cup U} \times \mathcal{L}_U\}, \\ \Sigma = \text{ the set of actions used in } CS, \\ s_0 = (v_0, P_0, C_0, 0) \end{array}$$

and the transition relation δ is the smallest transition relation satisfying the following. For all $(v, P, C, D) \in Q$, if x is the first clock in the sequence Y which is not in $aclk(P)$, and $(v, Act(P, x), C \wedge x = 0, D) \overset{a}{\hookrightarrow} (v', P', C', D')$ where C' is satisfiable, then we have: $\big((v, P, C, D), a, (v', P', prune_{X \setminus aclk(P')}(C'), D')\big) \in \delta$.

The transition relation $\hookrightarrow$ is specified by a set of rules, given in Fig. 6. Let us first explain these rules, after which we will go back to the explanation of Definition 4.4.

Synchronous Invocation. Rule *rSInv* states that a state $s = (v, sInv(S)_x, C, D)$ may evolve into the state $s' = (v', Stop, (x = t_S) \wedge C^{\uparrow}, D + t_S)$ via action $e \in \Sigma$, where *Stop* is the activity that does nothing, and t_S is the parametric response time of component service S. Note that, from Definition 4.4, the condition $(x = t_S) \wedge C^{\uparrow}$ is necessarily satisfied (otherwise this evolution is not possible). The resulting constraint is the intersection of constraints $C^{\uparrow}$ and $x = t_S$ (recall that the constraint $C^{\uparrow}$ denotes the time elapsing of C). Furthermore, the parametric duration from the initial state (D) is augmented with t_S. Rules *rSInv*, *rReply* and *rAInv* are similar.

Pick Activity. Rule *rPick*1 encodes the transition that takes place due to an *onMessage* activity. Let us explain the constraint $(x = t_S) \wedge\ idle(mpick_x) \wedge C^{\uparrow}$. First, after the transition, the current clock x needs to be equal to the parametric response time of service S, *i. e.*, $x = t_S$. Second, the constraint $idle(mpick_x)$ is added to ensure that x remains smaller or equal to the maximum duration of the $mpick_x$ activity. Third, the constraint $C^{\uparrow}$ denotes the time elapsing of C. Rule *rPick*2 is similar.

Conditional Activity. Given a conditional composition $A \lhd b \rhd B$, the guard condition b is a Boolean, hence its values are in $\{true, false\}$. As a consequence, given a valuation v of the variables, then $v(b) \in \{true, false, \bot\}$ (recall that $\bot$ denotes an uninitialized variable). We have that $v(b) = \bot$ when the evaluation of b is unknown, due to the fact that there may be uninitialized variables in b. Since b might be evaluated to either true or false at certain stages during runtime, we explore both activities A and B when $v(b) = \bot$ so as to reason about all possible scenarios. The case of $v(b) = \bot$ is captured by rules *rCond*1 and *rCond*2, and the cases where $v(b) \in \{true, false\}$ are captured by rules *rCond*3 and *rCond*4.

Sequential Activity. *rSeq*1 states that if activity A' is not a *Stop* activity (*i. e.*, activity A' has not finished its execution), then a state containing activity $A\,\mathbf{;}B$ may evolve into a state containing activity $A'\,\mathbf{;}B$. Otherwise, if A is a *Stop* activity (*i. e.*, activity A has finished its execution), then a state containing activity $A\,\mathbf{;}B$ may discharge activity A and evolve into a state containing B. This is captured by *rSeq*2.

Concurrent Activity. For concurrent activity $A \mid\mid\mid B$, either activity A or activity B can be executed. This is captured by *rFlow*1 and *rFlow*2 respectively. *rFlow*1 states that if state (v, A, C, D) can evolve into (v', A', C', D') via action $e \in \Sigma$, then a state containing $A \mid\mid\mid B$ can evolve into a state containing $A' \mid\mid\mid B$ via action $e \in \Sigma$, if $C' \wedge idle(B)$ holds. That is, the clock constraints in C' can exceed the duration activity B can last for. Rule *rFlow*2 is similar.

Let us now explain Definition 4.4. Starting from the initial state $s_0 = (v_0, P_0, C_0, 0)$, we iteratively construct successor states as follows. Given a state (v, P, C, D), a fresh clock x which is not currently associated with P is picked from Y. The state (v, P, C, D) is transformed into $(v, Act(P, x), C \wedge x = 0, D)$, *i. e.*, timed processes which just become activated are associated with x and C is conjuncted with $x = 0$. Then, a firing rule is applied to get a target state (v', P', C', D'). Lastly, clocks which do not appear within P' are pruned from C'. Observe that one clock is introduced and zero or more clocks may be pruned during a transition. In practice, a clock is introduced only if necessary; if the activation function does not activate any subprocess, no new clocks are created.

4.5. Good and Bad States

Let us defined good and bad states in the LTS obtained from Definition 4.4. The execution of a bad activity will make the execution of *CS* end in an undesired terminal

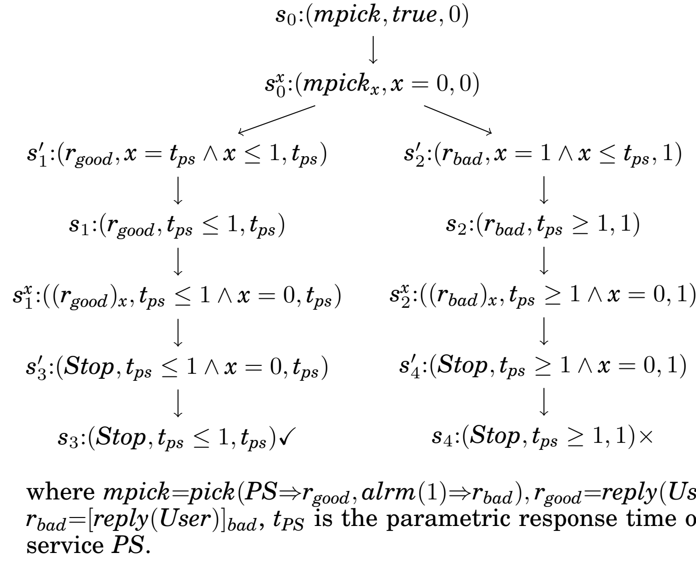


Fig. 7: Computing states of service CS (including intermediate states)

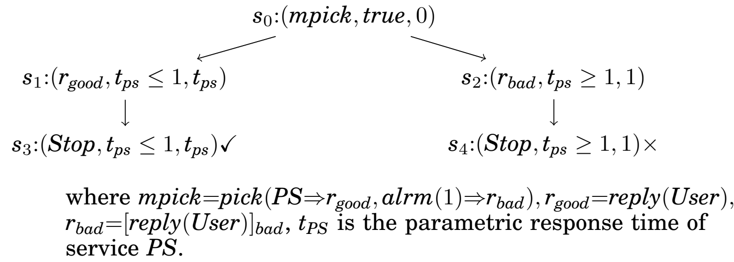


Fig. 8: LTS of service CS

state, which we refer to as a *bad state*. A terminal state which is not a bad state is called a *good state*. The synthesized local time requirement needs to guarantee the avoidance of all bad states and the termination of each run in a good state. The fact that each run must end in a good state is explained as follows: The non-determinism can be resolved at runtime depending on the variable values, or the response time of a component service. Therefore, we must guarantee that, regardless of the branch chosen by the composite service at runtime, it will end in a good state.

4.6. Application to an Example

Consider a composite service $CS = mpick$, where the definition of $mpick$ is given in Fig. 7. Noted that CS is a part of the SMIS example. The states of CS computed according to Definition 4.4 are given in Fig. 7, including intermediate states (detailed in the following). Since CS has no variable, $v = \varnothing$ in all states; therefore, we omit the component v from all states for the sake of brevity.

— At state s_0, the activation function assigns clock x to record time elapsing of pick activity $mpick$, with x initialized to zero. The tuple becomes the intermediate state $s_0^x = (mpick_x, x = 0, 0)$.

— From intermediate state s_0^x, the process may evolve into the intermediate state s'_1 by applying the rule $rPick1$, if the constraint $c_1 = ((x = t_{PS}) \wedge idle(mpick_x) \wedge (x = 0)^{\uparrow})$, where $idle(mpick_x) = (x \leq t_{PS} \wedge x \leq 1)$ and $(x = 0)^{\uparrow}$ (*i.e.*, $x \geq 0$), is satisfiable. Intuitively, c_1 denotes the constraint where t_{PS} time units elapsed since clock x has started. In fact, c_1 is satisfiable (for example with $t_{PS} = 0.5$ and $x = 0.5$). Therefore, it may evolve into the intermediate state $s'_1 = (r_{good}, (x = t_{PS}) \wedge idle(mpick_x) \wedge (x = 0)^{\uparrow}, t_{PS}) = (r_{good}, (x = t_{PS}) \wedge x \leq 1, t_{PS})$. Since clock x is not used anymore in $s'_1.P$ which is r_{good}, it is pruned. After pruning of clock variable x and simplification of the expression, the intermediate state s'_1 becomes the state $s_1 = (r_{good}, t_{PS} \leq 1, t_{PS})$.
— From intermediate state s_0^x, the process may evolve into the intermediate state s'_2, by applying the rule $rPick2$, if the constraint $c_2 = ((x = 1) \wedge idle(mpick_x) \wedge (x = 0)^{\uparrow})$, where $idle(mpick_x) = (x \leq t_{PS} \wedge x \leq 1)$ and $(x = 0)^{\uparrow}$ (*i.e.*, $x \geq 0$), is satisfiable. It is easy to see that c_2 is satisfiable; therefore, the process may evolve into the intermediate state $s'_2 = (r_{bad}, (x = 1) \wedge x \leq t_{PS}, 1)$. After clock pruning from intermediate state s'_2, it becomes state $s_2 = (r_{bad}, t_{PS} \geq 1, 1)$.
— From state s_1, activation function assigns clock x to the reply activity r_{good}, and the process evolves into intermediate state s_1^x. From intermediate state s_1^x, the process may evolve into intermediate state s'_3 by applying rule $rReply$, if the constraint $c_3 = ((x = 0) \wedge (t_{PS} \leq 1)^{\uparrow})$ is satisfiable, where $(t_{PS} \leq 1)^{\uparrow} = t_{PS} \leq 1$. In fact it is, and therefore it evolves into state $s'_3 = (Stop, t_{PS} \leq 1 \wedge (x = 0), t_{PS})$. After pruning of the non-active clock, it evolves into the terminal state $s_3 = (Stop, t_{PS} \leq 1, t_{PS})$. Since the terminal state is not caused by a bad activity, s_3 is considered as a good state, denoted by ✓ in Fig. 7.
— From state s_2, the process may also evolve into the terminal state $s_4 = (Stop, t_{PS} \geq 1, 1)$. Since the terminal state is caused by a bad activity, it is considered as a bad state, denoted by $\times$ in Fig. 7.

Note that all states s_i^x and s'_j, where $i, j \in \mathbb{N}$ and $0 \leq i \leq 4$, are intermediate states. State s_i^x is the state s_i after clock assignment operations are applied. State s'_j is the state s_j before clock pruning operations are applied. These intermediate states are given in Fig. 7 to illustrate in details the application of the semantics. The LTS of CS (without the intermediate states) is given in Fig. 8.

5. SYNTHESIZING THE STATIC LTC

Given $CS = (\mathcal{V}, U, P_0, C_0)$, the *global time requirement* for CS requires that, for every state (v, P, C, D) reachable from the initial state $(v_0, P_0, C_0, 0)$ in its LTS, the constraint $D \leq T_G$ is satisfied, where $T_G \in \mathbb{R}_{\geq 0}$ is the *global time constraint*. The *local time requirement* requires that if the response times of all component services of CS satisfy the *local time constraint* (LTC) $C_L \in \mathcal{C}_U$, then the service CS satisfies the global time requirement.

In this section, given a global time constraint T_G for a service CS, we present an approach to synthesize the static LTC (sLTC) C_L based on the LTS. The sLTC will be given in the form of an NNCC over U. We show that if the response times of all component services of CS satisfy the local time requirement, then the service CS will end in a good state within T_G time units.

5.1. Motivation

Assume a component service S. Assume that the only communication activity that communicates with S is the synchronous invocation activity $sInv(S)$. Upon invoking of service S, the activity $sInv(S)$ waits for the reply. The response time of S is equivalent to the waiting time in $sInv(S)$. Therefore, by analyzing the time spent in $sInv(S)$,

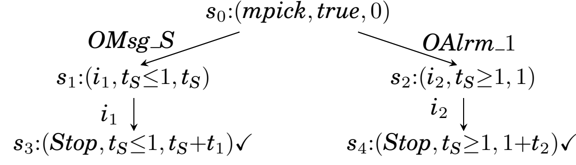


Fig. 9: LTS of composite service CS

we can get the response time of component service S. Given a composite service CS, let $t_i \in \mathbb{R}_{\geq 0}$ be the response time of component service S_i for $i \in \{1, \ldots, n\}$, and let $E_t = \{t_1, \ldots, t_n\}$ be a set of component service response times that fulfill the global time requirement of service CS. Because t_i, for $i \in \{1, \ldots, n\}$, is a real number, there are infinitely many possible values, even in a bounded interval (and even if one restricts these values to rational numbers). A method to tackle this problem is to reason *parametrically*, by considering these response times as unknown constants, or *parameters*. Let $u_i \in \mathbb{Q}_{\geq 0}$ be the parametric response time of component service S_i for $i \in \{1, \ldots, n\}$, and let $E_u = \{u_1, \ldots, u_n\}$ be the set of component service parametric response times. Using constraints on E_u, we can represent an infinite number of possible response times symbolically. The local time requirement of component services of CS is specified as a constraint over E_u. An example of a local time requirement is $(u_1 \leq 6) \wedge (u_2 \leq 5)$. This local time requirement specifies that, in order for CS to satisfy the global time requirement, service S_1 needs to respond within 6 time units, and service S_2 needs to respond within 5 time units. A local time requirement can also be in the form of a dependency between parametric response times, *e.g.*, $(u_2 \leq u_1 \Rightarrow u_1 + u_2 \leq 6) \wedge (u_1 \leq u_2 \Rightarrow u_1 \leq 6)$. This example requires that when service S_2 responds not slower than service S_1, then the sum of the response times of services S_1 and S_2 must be at most 6 seconds; however, if service S_1 responds not slower than service S_2, then service S_1 must respond within 6 seconds.

5.2. Addressing the Good States

We assume a composite service CS and its LTS $LTS_{CS} = (Q, s_0, \Sigma, \delta)$; let Q_{good} be the set of all good states of service LTS_{CS}. In this section, we assume there are no bad states; we will discuss bad states in Section 5.3.

Given LTS_{CS}, our goal is to synthesize the local time requirement for service CS. We make two observations here. First, a good state $s_g = (v_g, P_g, C_g, D_g) \in Q_{good}$ is reachable from the initial state s_0 iff C_g is satisfiable. Second, whenever the good state s_g is reached, we require that the total delay from initial state s_0 to state s_g must be no larger than the global time constraint T_G, *i.e.*, $D_g \leq T_G$. To sum up, given a good state $s_g = (v_g, P_g, C_g, D_g)$ where $s_g \in Q_{good}$, we require the constraint $(C_g{\downarrow_U} \Rightarrow (D_g \leq T_G))$ to hold. The constraint means that whenever s_g is reachable from s_0, the total (parametric) delay from s_0 to s_g must be less than the global time constraint T_G. The synthesized sLTC for CS is the conjunction of such constraints for each good state $s_g \in Q_{good}$, that is:

$$\bigwedge_{(v_g, P_g, C_g, D_g) \in Q_{good}} (C_g{\downarrow_U} \Rightarrow (D_g \leq T_G)).$$

Example. Let us consider a composite service CS whose process component is $pick(S \Rightarrow i_1, alrm(1) \Rightarrow i_2)$ (henceforth referred to as $mpick$), where S is a component service. Suppose the global time requirement of the composite service CS is to respond within five seconds. Fig. 9 shows the LTS of CS, where i_j denotes $sInv(S_j)$, such that S_j is a component service with parametric response time t_j, for $j \in \{1, 2\}$.

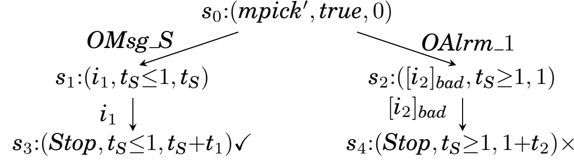


Fig. 10: LTS of composite service CS'

For composite service CS in Fig. 9, we have two good states (states s_3 and s_4), and the synthesized local time requirement for composite service CS is:

$$(t_S \leq 1) \Rightarrow (t_S + t_1 \leq 5) \wedge (t_S \geq 1) \Rightarrow (1 + t_2 \leq 5)$$

5.3. Addressing the Bad States

Another goal we want to achieve is to avoid all bad states in LTS_{CS}. Let Q_{bad} be the set of all bad states of service LTS_{CS}. Given a bad state $s_b = (v_b, P_b, C_b, D_b) \in Q_{bad}$, this bad state must not be reachable from the initial state s_0. Hence, in order to prevent C_b to be satisfiable, we require that the parameters be taken in the negation of the projection of C_b onto U, *i. e.*, we require that $\neg(C_b)\downarrow_U$ be satisfiable. The synthesized sLTC for CS is the conjunction of such constraints for each bad state $s_b \in Q_{bad}$, that is:

$$\bigwedge_{(v_b, P_b, C_b, D_b) \in Q_{bad}} \big(\neg(C_b)\downarrow_U\big).$$

Example. Consider a variant of the example in Fig. 9, where the definitions of i_1 and i_2 remain the same. But now, i_2 is treated as a bad activity, which is represented as $[i_2]_{bad}$. That is, the composite service becomes a composite service CS' whose process component is $pick(S \Rightarrow i_1, alrm(1) \Rightarrow [i_2]_{bad})$ (henceforth referred to as $mpick'$). This service results in the LTS shown in Fig. 10, where state s_4 is a bad state. We use this example to provide the intuition how to modify the synthesized NNCC to avoid reaching bad states. Note that the constraint $s_4.C = t_S \geq 1$ is introduced by the *pick* activity. A way to avoid the reachability of s_4 is to prevent the transition $OAlrm_1$ from firing. An effective way to achieve this is by adding the negation $\neg\ (s_4.C\downarrow_U)$ to the synthesized NNCC. Therefore, the local time requirement for composite service CS' is $(s_3.C\downarrow_U \Rightarrow (s_3.D \leq T_G)) \wedge \neg\ (s_4.C\downarrow_U)$. This NNCC can ensure that any complete run of the service ends in a good state. (This will be proved in Section 5.6.)

5.4. Synthesis Algorithms

Algorithm 1 presents the entry algorithm for synthesizing the sLTC for a given service CS, by traversing the LTS (Q, s_0, Σ, δ) of CS. Algorithm 1 simply calls synthLTS(s) applied to the initial state s_0; this second algorithm is given in Algorithm 2.

Algorithm 1: synthSLTC(CS)

input : Composite service model CS with initial state s_0
output: The sLTC $C_L \in \mathcal{NC}_U$

1 **return** synthLTS(s_0);

Given a state $s = (v, P, C, D)$ in the LTS of service CS, synthLTS(s) returns a constraint $C \in \mathcal{C}_U$. If state s is a good state (line 1), then it returns the constraint

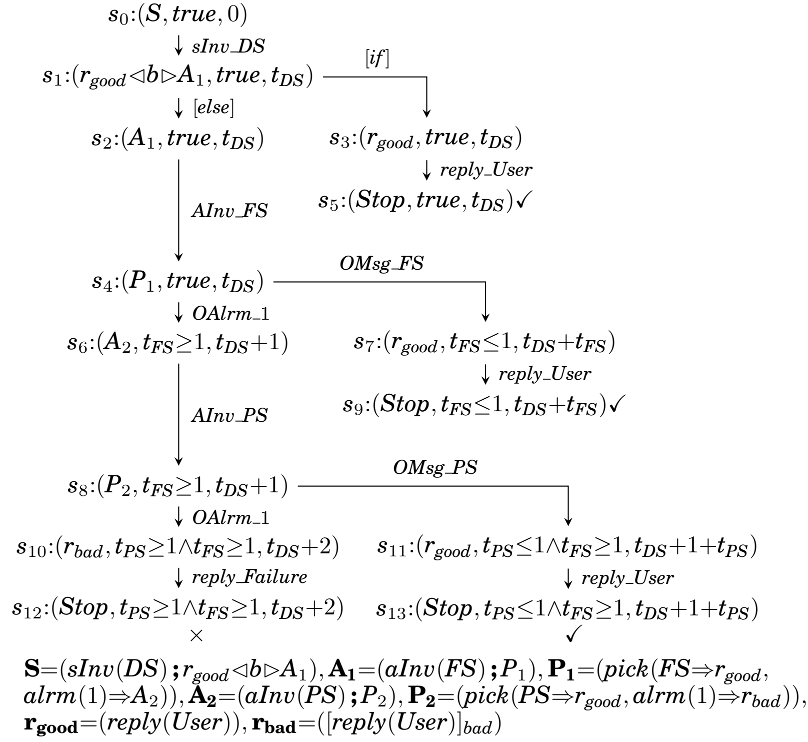


Fig. 11: LTS of the SMIS

$s.C{\downarrow_U} \Rightarrow (s.D \leq T_G)$ (line 2), where T_G is the given global time constraint of the service CS. If state s is a bad state (line 3), then the negation of the current constraint $s.C{\downarrow_U}$ is returned (line 4). If s is a non-terminal state (line 5), the algorithm returns the conjunction of the result of the algorithm recursively applied on the successors of s (line 6).

Algorithm 2: synthLTS(s)

input : State s of LTS
output: The constraint for LTS that starts at s

```
1 if s is a good state then
2     return (s.C↓U ⇒ (s.D ≤ TG));
3 else if s is a bad state then
4     return ¬ (s.C↓U);
5 else
      // s is a non-terminal state
6     return ⋀_{s'∈succ(s)} synthLTS(s') ;
```

5.5. Application to the Running Example

Fig. 11 shows the LTS of the running example introduced in Section 3. Algorithm synthLTS(s) is used to synthesize the local time requirement for SMIS based on the

$(t_{DS} \leq 3) \wedge (t_{FS} \leq 1) \Rightarrow (t_{DS}+t_{FS} \leq 3) \wedge$
$(t_{FS} \geq 1 \wedge t_{PS} \leq 1) \Rightarrow (t_{DS}+t_{PS} \leq 2) \wedge \neg (t_{FS} \geq 1 \wedge t_{PS} \geq 1)$

Fig. 12: sLTC of SMIS

$(t_{FS} < 1 \wedge t_{DS} + t_{FS} \leq 3) \vee (t_{PS} < 1 \wedge t_{FS} > 1 \wedge t_{DS} + t_{PS} \leq 2) \vee$
$(t_{PS} < 1 \wedge t_{DS} + t_{FS} \leq 3 \wedge t_{DS} + t_{PS} \leq 2)$

Fig. 13: sLTC of SMIS after simplication

LTS. The sLTC of the running example is shown in Fig. 12. After simplification[3] using Z3 [de Moura and Bjørner 2008], a Satisfiability Modulo Theories (SMT) solver developed by Microsoft Research, we get the sLTC shown in Fig. 13. We first translate our expressions into Z3 expressions, and apply Z3 built-in strategies (*e. g.*, simplify) to get the equivalent DNF formulas.

This result provides us useful information on how the component services collectively satisfy the global time constraint. That is useful when selecting component services. For the case of SMIS, one way to fulfill the global time requirement of SMIS is to select component service FS with response time that is less than 1 second, and component services DS and FS where the summation of their response times should be less than or equal to 3 seconds.

Service Selection. Recall that the *stipulated response time* of a component service S denotes the upper bound on its response time with respect to the synthesized constraint. The sLTC can be used to select a set of services that collectively satisfy the global time requirement of a composite service. Given a composite service CS with n component services $E = \{S_1, S_2, \ldots, S_n\}$, let $\{t_1, t_2, \ldots, t_n\}$ and $\{st_1, st_2, \ldots, st_n\}$, where $t_i \in \mathcal{L}_U$ and $st_i \in \mathbb{R}_{\geq 0}$ are the parametric response times and stipulated response times for component services in E respectively. One can check whether the component services can collectively satisfy the sLTC of the composite service CS, by checking the satisfiability of the formula $(\bigwedge_{1 \leq i \leq n} t_i \leq st_i) \Rightarrow \mathsf{synthSLTC}(CS)$.

For the SMIS example, we have $(t_{FS} \leq 1.5 \wedge t_{PS} \leq 1.5 \wedge t_{DS} \leq 0.5) \Rightarrow \mathsf{synthSLTC}(SMIS)$. This means that if we select component services FS, PS, and DS that respond within 1.5 seconds, 1.5 seconds, 0.5 seconds respectively, we always guarantee the sLTC of SMIS.

5.6. Termination and Soundness

We show the termination and soundness of synthesis of synthSLTC.

5.6.1. Termination

LEMMA 5.1. *Let CS be a service model. Then LTS_{CS} is acyclic and finite.*

PROOF. *From Assumption 1 and from the fact that there are no recursive activities in BPEL.* □

PROPOSITION 5.2. *Let CS be a service model. Then* synthSLTC(CS) *terminates.*

PROOF. *From Lemma 5.1, LTS_{CS} is acyclic. Algorithm 1 is obviously non-recursive. Now, Algorithm 2 is recursive (line 6). However, due to the acyclic nature of LTS_{CS} and*

[3]For readability, we give the constraint as output in disjunctive normal form (DNF), instead of the usual conjunctive normal form (CNF).

the fact that Algorithm 2 is called recursively on the successors of the current state, then no state is explored more than once. This ensures termination. □

5.6.2. Soundness. In this section, we prove that for any parameter valuation satisfying the output of synthSLTC, any complete run ends in a good state, and all reachable good states are reachable within the global delay T_G.

We first need several definitions and intermediate results. Given a parametric service model *CS* and a parameter valuation π, let us relate runs of LTS_{CS} and $LTS_{CS[\pi]}$.

Definition 5.3 (*equivalent runs*). Let *CS* be a parametric service model, and let π be a parameter valuation.

Let $\rho = \langle(v_0,P_0,C_0,D_0),a_0,(v_1,P_1,C_1,D_1),\dots,a_{n-1},(v_n,P_n,C_n,D_n)\rangle$ be a run of $LTS_{CS[\pi]}$. Let $\rho' = \langle(v'_0,P'_0,C'_0,D'_0),a'_0,(v'_1,P'_1,C'_1,D'_1),\dots,a'_{n-1},(v'_n,P'_n,C'_n,D'_n)\rangle$ be a run of LTS_{CS}.

The two runs ρ and ρ' are *equivalent* if $v_i = v'_i$ and $P_i = P'_i[\pi]$ for $0 \leq i \leq n$ and $a_i = a'_i$ for $0 \leq i \leq n-1$.

The following lemma states that, given a run of $LTS_{CS[\pi]}$, there exists a unique equivalent run in LTS_{CS}. Since we defined runs as alternating symbolic states and *actions* (and not, more generally, edges) the uniqueness can only be proven if the LTS is deterministic, *i. e.*, if from a given state, all outgoing actions are pairwise different. This is not true if, in the composite service, different calls are made to the same service from the same process. As a consequence, without loss of generality, we assume that actions are made different if necessary, by labeling with different actions different calls: if two different calls to the same service can be made from the same process in different program locations (*e. g.*, two asynchronous invocations *AInv_FS*), then we label them differently in the LTS (*e. g.*, *AInv_FS*1 and *AInv_FS*2 respectively).

LEMMA 5.4. *Let CS be a parametric service model, and let π be a parameter valuation. Let ρ_π be a run of $LTS_{CS[\pi]}$.*
Then there exists a unique run of LTS_{CS} equivalent to ρ_π.

PROOF. By induction on the length of the runs. We prove in fact a slightly stronger result: given a state (v,P,C,D) of a run ρ in $LTS_{CS[\pi]}$, and given a state (v',P',C',D') of the equivalent run ρ' in LTS_{CS}, we show that these states are not only equivalent, but also that $C \subseteq C'$.

Base case. From Definition 4.4, the initial state of LTS_{CS} is $(v_0,P_0,C_0,0)$. The initial state of $LTS_{CS[\pi]}$ is $(v_0,P_0[\pi],C_0[\pi],0)$. Since $C_0[\pi] \subseteq C_0$, then the result trivially holds.

Induction step. Assume ρ_π is a run of $LTS_{CS[\pi]}$ of length m reaching state (v_1,P_1,C_1,D_1); assume there exists a unique run of LTS_{CS} equivalent to ρ_π and of length m, reaching state (v'_1,P'_1,C'_1,D'_1). From Definition 5.3, it holds that $v_1 = v'_1$ and $P_1 = P'_1[\pi]$. From the induction hypothesis, it holds that $C_1 \subseteq C'_1$.

Let (v_2,P_2,C_2,D_2) be the successor state of (v_1,P_1,C_1,D_1) via action a in ρ_π.

Assume (v_2,P_2,C_2,D_2) is obtained from (v_1,P_1,C_1,D_1) by applying rule *rSInv* in Fig. 6. Since $C_1 \subseteq C'_1$, then rule *rSInv* can also be applied to (v'_1,P'_1,C'_1,D'_1), yielding a state (v'_2,P'_2,C'_2,D'_2). Now, we have:

$C_1 \subseteq C'_1 \implies C_1^{\uparrow} \subseteq C_1'^{\uparrow}$
$\implies (x = t_S \wedge C_1^{\uparrow}) \subseteq (x = t_S \wedge C_1'^{\uparrow})$
$\implies C_2 \subseteq C'_2$.

In particular, $C_2 \subseteq C'_2$ implies that C'_2 is non-empty, hence the state (v'_2,P'_2,C'_2,D'_2) is a valid state. In addition, since $P_1 = P'_1[\pi]$ and rule *rSInv* derives to *Stop*, then

$P_2 = P'_2[\pi]$. Similarly, variables are updated in the same manner on both sides, hence $v_2 = v'_2$. The proof is similar for other rules in Fig. 6.
Finally, thanks to the hypothesis of determinism on action labels, then the successor state (v'_2, P'_2, C'_2, D'_2) is the unique successor state of (v'_1, P'_1, C'_1, D'_1) in LTS_{CS} via action a. Hence there exists a unique run of LTS_{CS} equivalent to ρ_π and of length $m + 1$.

□

In the following, given a run ρ_π of $LTS_{CS[\pi]}$, from Lemma 5.4 we can safely refer to the run of LTS_{CS} equivalent to ρ_π.

Now, we define a state of an LTS $LTS_{CS[\pi]}$ as π-(non-)terminal if there exists an equivalent (non-)terminal state in LTS_{CS}. Formally:

Definition 5.5 (*π-terminal state*). Let CS be a parametric service model, and let π be a parameter valuation, Let ρ_π be a run of $LTS_{CS[\pi]}$ reaching a state s_π.

Let ρ be the run of LTS_{CS} equivalent to ρ_π, and reaching a state s.

s_π is a π-*non-terminal* (resp. π-*terminal*) state if s is non-terminal (resp. terminal) in LTS_{CS}.

Given $CS[\pi]$, we now introduce the notion of early-deadlocked LTS: $LTS_{CS[\pi]}$ is early-deadlocked if it contains a state that is a terminal state in $LTS_{CS[\pi]}$, although its equivalent state in LTS_{CS} is non-terminal. Formally:

Definition 5.6 (*early-deadlocked LTS*). Let CS be a parametric service model, let π be a parameter valuation.

The LTS of $CS[\pi]$ is *early-deadlocked* if it contains at least one state s such that s is a π-non-terminal state, and s is a terminal state in $LTS_{CS[\pi]}$.

This happens because $\pi \not\models s'.C$, for each $s' \in succ(s)$ in the LTS of parametric service model CS.

In the following Lemma 5.7, we show that, given $\pi \models \mathsf{synthSLTC}(CS)$, then $CS[\pi]$ is not early-deadlocked. Note that this does not mean that the system is deadlock-free; on the contrary, the system always eventually ends in a deadlock situation, since all terminal states have no successor (we consider acyclic systems). However, we show that all terminal states of $LTS_{CS[\pi]}$ are also π-terminal states.

LEMMA 5.7. *Let CS be a parametric service model. Let* $\pi \models \mathsf{synthSLTC}(CS)$. *Then* $LTS_{CS[\pi]}$ *is not early-deadlocked.*

PROOF. *We reason here on the traversal of the LTS of CS, as* synthSLTC *relies on this traversal. Assume* $LTS_{CS} = (Q, s_0, \Sigma, \delta)$. *The constraint associated with the initial state is true, i. e.,* $s_0.C = true$, *therefore it is always satisfiable. Given a state s and a state* s' *such that* $s' \in succ(s)$, *the situation where* $\pi \models s.C$ *and* $\pi \not\models s'.C$ *can only happen when* $s'.C{\downarrow_U}$ *is stronger than* $s.C{\downarrow_U}$, *i. e.,* $s'.C{\downarrow_U} \subsetneq s.C{\downarrow_U}$. *In such a case, the additional constraints in* $s'.C{\downarrow_U}$ *could only be introduced by a pick or a flow activity using the idle function (see Fig. 6), for the purpose of constraining the relative speed of the services. Assume the pick construct* $mpick = pick(S{\Rightarrow}P, alrm(a){\Rightarrow}Q)$, *where S is a service with parameter response time* t_S, $a \in \mathbb{R}_{\geq 0}$ *and* P, Q *are composite service activities. For the activity P to be enabled, the satisfaction of constraint* $t_S \leq a$ *is required, while for the activity Q to be enabled, the satisfaction of constraint* $t_S \geq a$ *is required. Since given any parameter valuation* π, *mpick will be able to execute either of the branches, therefore it cannot be deadlocked. Assume the concurrent activity as* $conc = P \mid\mid\mid Q$, *where* P, Q *are composite service activities. If P (resp. Q) is a reply or an asynchronous invocation activity, then P (resp. Q) is always executable, since it takes no time. If P and Q are*

synchronous invocation or receive activities, which takes parameter response time t_P and t_Q respectively, then activity P is executable, if $t_P \leq t_Q$, and activity Q is executable if $t_Q \leq t_P$. Since given any parameter valuation π, either of the branches in conc is executable, therefore it cannot be deadlocked. □

The following lemmas will be used to prove the subsequent Theorem 5.11.

LEMMA 5.8. *Let CS be a service model. Let $\pi \models \mathsf{synthSLTC}(CS)$. Then no bad state is reachable in $LTS_{CS[\pi]}$.*

PROOF. *Let $K = \mathsf{synthSLTC}(CS)$. K is a conjunction of "good" parameter constraints (accumulated from line 2 in Algorithm 2) and "bad" parameter constraints (accumulated from line 4 in Algorithm 2). Hence, K contains at least the negated constraints of all bad states. Hence, the bad states are unreachable for any $\pi \models K$.* □

LEMMA 5.9. *Let CS be a service model. Let $\pi \models \mathsf{synthSLTC}(CS)$. Then any complete run of $LTS_{CS[\pi]}$ ends in a good state.*

PROOF. *First, note that the initial state s_0 is reachable in $LTS_{CS[\pi]}$ (since $s_0.C = true$). If the initial state is the only state, then from Lemma 5.8, it is also not a bad state; hence it is a good state. Now, if it is not the only state, from the absence of intermediate deadlocks (Lemma 5.7), from the finiteness of the LTS (Lemma 5.1) and from the absence of bad states (Lemma 5.8), then any run of $LTS_{CS[\pi]}$ ends in a good state.* □

LEMMA 5.10. *Let CS be a service model. Let $\pi \models \mathsf{synthSLTC}(CS)$. Then for all good state (v, P_g, C, d) of $LTS_{CS[\pi]}$, $d \leq T_G$.*

PROOF. *Let $s_g = (v, P_g, C, D)$ be a reachable state in LTS_{CS} such that s_g is a good state. From Definition 4.4, C is satisfiable (and hence $C\downarrow_U$ too). Since s_g is a good state, Algorithm* synthLTS *added a constraint $C\downarrow_U \Rightarrow D \leq T_G$ to the result. Hence, $\mathsf{synthSLTC}(CS) \subseteq (C\downarrow_U \Rightarrow D \leq T_G)$. Now, for any $\pi \models \mathsf{synthSLTC}(CS)$, we have that $\pi \models (C\downarrow_U \Rightarrow D \leq T_G)$, and hence all reachable states in $LTS_{CS[\pi]}$ are such that $d \leq T_G$.* □

We can now formally state the soundness of synthSLTC.

THEOREM 5.11. *Let CS be a service model. Let $\pi \models \mathsf{synthSLTC}(CS)$. Then:*

(1) *Any complete run of $LTS_{CS[\pi]}$ ends in a good state.*
(2) *For all good state (v, P_g, C, d) of $LTS_{CS[\pi]}$, $d \leq T_G$.*

PROOF. *From Lemmas 5.9 and 5.10.* □

Given a composite service CS, and assume $S_g = \{s_1, \ldots, s_n\}$ be the set of all good states in LTS_{CS}. In the following proposition, we show that any $\pi \models \mathsf{synthSLTC}(CS)$ necessarily satisfies (at least) one of the good states' constraints, *i.e.*, $\pi \models s_i.C\downarrow_U$ for some $s_i \in S_g$. Indeed, recall $\mathsf{synthSLTC}(CS)$ is a conjunction of good and bad constraints. In the following proposition, we show that the good constraints of the form $P = (c_1 \Rightarrow r_1 \wedge \ldots \wedge c_n \Rightarrow r_n)$ will not hold trivially by just having $c_i = false$, for all $i \in \{1, \ldots, n\}$.

PROPOSITION 5.12. *Let CS be a service model, and Q_{good} be the set of all good states in LTS_{CS}. Let $\pi \models \mathsf{synthSLTC}(CS)$.*
Then $\exists s \in Q_{good} : \pi \models s.C\downarrow_U$.

PROOF. *From Algorithm 2, $\mathsf{synthSLTC}(CS)$ is a conjunction of "good" constraints (accumulated from line 2 in Algorithm 2) and "bad" constraints (accumulated from line 4 in Algorithm 2). That is, assume $\mathsf{synthSLTC}(CS) = (C_g \wedge C_b)$, where $C_g =$*

$\bigwedge_{s_i \in Q_{good}} (s_i.C{\downarrow_U} \Rightarrow (s_i.D \leq T_G))$, *and* T_G *be the global time constraint, and* $C_b = \bigwedge_{s_j \in Q_{bad}} \neg(s.C_j{\downarrow_U})$. *Hence, since* $\pi \models \mathsf{synthSLTC}(CS)$ *then* $\pi \models C_g$, *hence* $\exists s \in Q_{good}$: $\pi \models s.C{\downarrow_U}$. □

5.7. Incompleteness of synthSLTC

A limitation of synthSLTC is that it is incomplete, *i. e.*, it does not include all parameter valuations that could give a solution to the problem of the local time requirement. Given an expression $A \triangleleft \mathrm{a{=}1} \triangleright B$, since a may be unknown at design time, we explore both branches (activities A and B) for synthesizing the sLTC. Nevertheless, only exactly one of these activities will be executed at runtime. Including constraints from activities A and B will make the constraints stricter than necessary; therefore some of the feasible parameter valuations are excluded – this makes the synthesis by synthSLTC incomplete. This can be seen as a trade-off to make the synthesized local time requirement more general, *i. e.*, to hold in any composite service instance. In Section 6, we will introduce a method that leverages on runtime information to mitigate this problem.

6. RUNTIME REFINEMENT OF LOCAL TIME REQUIREMENT

In order to improve the local time requirement computed statically using the algorithms presented in Section 5, we introduce in this section a *refined* local time requirement, together with its usage for runtime adaptation of a service composition.

6.1. Motivation

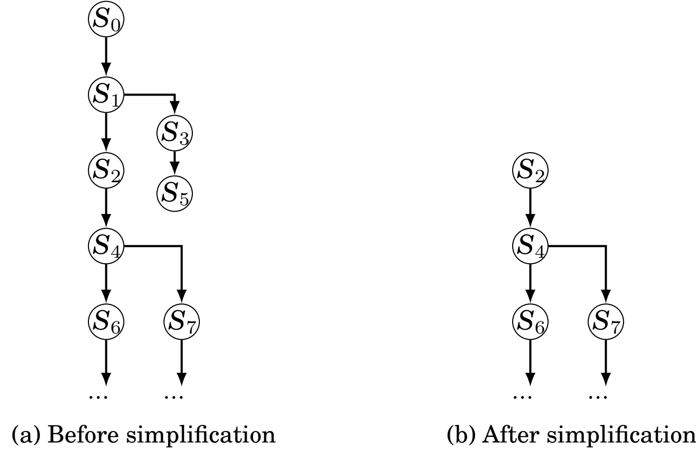
Let us consider a composite service CS. Assume that we have selected a set of component services such that their stipulated response times fulfill the sLTC of CS. Since the composite service is executed under a highly evolving dynamic environment, the design time assumptions may evolve at runtime. For instance, the response times of component services could be affected by network congestion. This may result in the non-conformance of stipulated response times for some component services. However, the non-conformance of stipulated response times of component services does not necessary imply that the composite service will not satisfy its global time requirement. This is because the sLTC is synthesized at the design time to hold in *any* execution trace of CS; whereas at runtime, the runtime information can be used to synthesize a more relaxed constraint for CS.

More specifically, given a composite service CS, we have two pieces of runtime information that may help to synthesize a more relaxed constraint: the execution path that has been taken by CS, and the elapsed time of CS. First, the execution path taken by CS can be used for LTS simplification. This is because in the midst of execution, some of the execution traces can be disregarded and therefore a weaker LTC, that includes more parameter valuations, may be synthesized. Second, the time elapsed of CS can be used to instantiate some of the response time parameters with real time constants; this makes the synthesized LTC contain less uncertainty and be more precise.

For example, consider the SMIS composite service, the LTS of which is depicted in Fig. 11. At runtime, after invocation of the component service DS, SMIS will be at state s_2. Assume that DS does not conform to its stipulated response time. Therefore, it is desirable to check whether invoking FS can still satisfy the global time requirement of CS. One can make use of sLTC for this purpose. Nevertheless, a more precise LTC may be synthesized at state s_2.

The first observation is that, from state s_2, we can safely ignore the constraints from the good state s_5, since it is not reachable from s_2. The second observation is that the delay from state s_0 to state s_2 (say r seconds, with $r \in \mathbb{R}_{\geq 0}$) is known. For this reason,

we can substitute the delay component of state s_2, which is the parametric response time t_{DS}, with the actual time delay r. This motivates the use of runtime information of the composite service to refine the LTC. We refer to the runtime refined LTC as the runtime LTC (denoted by rLTC). In addition to this refinement, we can also simplify the LTS by pruning the states corresponding to past states (*e. g.*, s_0, s_1 in Fig. 11), as well as the successors of these past states that were not met in practice (*e. g.*, s_3 and s_5 in Fig. 11), because another branch was taken at runtime. We show the LTS of SMIS before and after simplification in Figs. 14a and 14b respectively.



(a) Before simplification (b) After simplification

Fig. 14: LTS Simplification of SMIS

By incorporating the runtime information, the resulting rLTC at state s_2 is:

$$(t_{FS} \leq 1) \Rightarrow (r + t_{FS} \leq 3) \land$$
$$(t_{FS} \geq 1 \land t_{PS} \leq 1) \Rightarrow (r + t_{PS} \leq 2) \land$$
$$\neg\, (t_{FS} \geq 1 \land t_{PS} \geq 1)$$

Although the synthesized rLTC is still incomplete, nevertheless by incorporating runtime information, it allows synthesizing a constraint weaker than sLTC. By allowing more parameter valuations, rLTC mitigates the problem of the incompleteness of the sLTC.

6.2. Runtime Adaptation of a BPEL Process

In this section, we introduce a service adaptation framework to improve the conformance of global time requirement for a composite service. The framework makes use of rLTC and the architecture of the framework is shown in Fig. 15. There are two modules in the framework – Runtime Engine Module (*RE*) and Service Monitoring Module (*SM*). The Runtime Engine Module (*RE*) provides an environment for the execution of a BPEL service; here, we use ApacheODE [Foundation 2007], an open source BPEL engine. We instrument the runtime component of Apache ODE to communicate with the service monitoring module.

The Service Monitoring Module (*SM*) is used to monitor the execution of a BPEL service. During the deployment of a service *CS*, *SM* generates the LTS (Q, s_0, Σ, δ) of *CS* and stores it in the cache of *SM* so that it is available when *CS* is executing.

During the execution of the composite service *CS*, the executed action $a_e \in \Sigma$ from *RE* is used to update the active state $s_a \in Q$ of LTS stored in *SM*. The action a_e is also

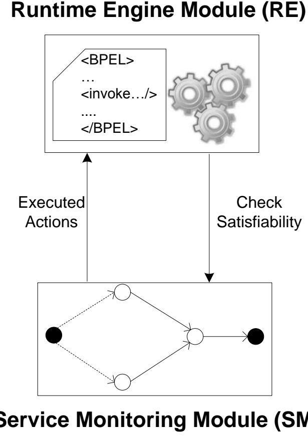


Fig. 15: Service adaptation framework

stored as part of the current execution run. *SM* also keeps track of the total execution time for this execution run, as well as the response time for each component service invocation.

Prior to the invocation of a component service S, *RE* will consult *SM* to check the satisfiability of rLTC. If the rLTC of s_a is satisfiable, then *SM* will instruct *RE* to continue invoking S as usual. Otherwise, some kind of mitigation procedure may be triggered. One of the possible mitigation procedures is to invoke a backup service of S, S_{bak}, which has a faster stipulated response time than S (that may come with a cost). An example of CS, S and S_{bak}, are services SMIS, *FS* and *PS* respectively.

In the following, we introduce the details on the synthesis of rLTC (Section 6.3) and satisfiability checking (Section 6.4).

6.3. Algorithm for Runtime Refinement

A way to calculate the rLTC could be to run synthSLTC (Algorithm 2) from a state s in the LTS. However, this requires traversing the state-space repeatedly for every calculation of the rLTC. To make it more efficient, we extend synthSLTC by calculating the rLTC for each state s during the synthesis of the LTC at the design time. Therefore, at runtime, we only need to retrieve the synthesized rLTC of the corresponding state for direct usage.

synthRLTC (given in Algorithm 3) synthesizes the rLTC for each state in the LTS. Before explaining the algorithm, let us introduce a few notations used in Algorithm 3. First, we assume that states in the LTS of CS are augmented with an additional "field" to store the computed rLTC. We use $s.rLTC$ to denote the $rLTC$ associated with state s. Additionally, we use the following shorthand to perform a conjunction of pairs of parametric constraints $(cons_i.g, cons_i.b)$ such that the resulting pair is such that its left-hand (resp. right-hand) side is the conjunction of all left-hand (resp. right-hand) sides: $\bigsqcap\big((cons_1.g, cons_1.b), \ldots, (cons_n.g, cons_n.b)\big)$ denotes $\big((cons_n.g \wedge \ldots \wedge cons_n.g), (cons_n.b \wedge \ldots \wedge cons_n.b)\big)$.

Given a composite service CS together with its associated LTS, and a state in LTS_{CS}, synthRLTC returns a constraint pair $c_s = (g, b)$, where $g, b \in \mathcal{C}_U$. In this pair, g (resp. b) denotes the constraint associated to a good (resp. bad) state. Given a constraint pair c_s, we use $c_s.g$ (resp. $c_s.b$) to refer to the first (resp. second) component of c_s. Variables d_f and r_f are *free variables*, which are variables to be substituted at runtime. In par-

ticular, given a state s, free variables d_f and r_f in $s.rLTC$ are to be substituted by the delay component $s.D \in \mathcal{L}_U$ and the actual delay $r \in \mathbb{R}_{\geq 0}$ from the initial state to the state s respectively.

Algorithm 3: synthRLTC(CS, LTS_{CS}, s)

input : Composite service CS
input : LTS LTS_{CS} of CS
input : State s in LTS of CS
output: Constraint pair for sub-LTS of CS starting with s

```
1  cons ← ∅;
2  if s is a good state then
3  |   cons ← (s.C↓_U ⇒ (s.D − d_f + r_f ≤ T_G), true);
4  |   s.rLTC ← cons.g ∧ (d_f = s.D);
5  else if s is a bad state then
6  |   cons ← (true, ¬ (s.C↓_U));
7  |   s.rLTC ← cons.b;
8  else
   |   // s is a non-terminal state
9  |   cons ← ⨅_{s′∈succ(s)} synthRLTC(s′);
10 |   s.rLTC ← cons.g ∧ cons.b ∧ (d_f = s.D);
11 return cons;
```

Let us now explain synthRLTC in details. Given a good state s (line 2), $s.rLTC$ is assigned with value $cons.g$, with free variable d_f substituted with $s.D$ (line 4); note that substitution is here achieved using conjunction of the constraint with the equality $d_f = s.D$. As an illustration, consider the good state s_{13} in the SMIS example (the LTS of which is given in Fig. 11). At runtime, assume the active state is at state s_{13}, and assume that it takes $r \in \mathbb{R}_{\geq 0}$ seconds to execute from the initial state s_0 to state s_{13}. Therefore, the previously unknown parametric response time in the delay component of state s_{13}, *i. e.*, $t_{DS} + 1 + t_{PS}$, can be substituted with the real value r. To achieve this, at line 3, we subtract away the free variable d_f, which is to be substituted with the response time parameter of state s_{13}, and add back the free variable r_f, which is to be substituted with the real value r. We substitute the free variable d_f at line 4. For free variable r_f, it is only substituted in Algorithm 4 at runtime when the delay is known. In the case of the SMIS example, the $rLTC$ of state s_{13} after substituting free variable r_f with value r (*i. e.*, $s_{13}.rLTC \wedge (r_f = r)$) is $((t_{PS} \leq 1 \wedge t_{FS} \geq 2) \Rightarrow (r \leq 3))$.

When s is a bad state (lines 5 to 7), we simply compute the negation of the associated constraint so as to keep the system reaching this bad state (just as in Algorithm 2).

When s is a non-terminal state (line 8), $s.rLTC$ is assigned with the conjunction of all good and bad constraints computed by recursively calling synthRLTC on the successor states of s, where free variable d_f is substituted with $s.D$ (line 10). The reason for taking the conjunction of both good and bad constraints is to guarantee any complete run from state s ends in a good state, and to avoid the reachability of any bad state from s. Also note that the rLTC of the initial state s_0 is the same as its sLTC, *i. e.*, $s_0.rLTC = sLTC$; the reason is that at the initial state s_0, there is no runtime information for refining the sLTC, hence the refined LTC is equal to the static LTC. In fact, one can see Algorithm 3 as a generalization of Algorithm 1, in the sense that Algorithm 3

can be applied to any state (not only the initial one), and can benefit from the current partial execution.

6.4. Satisfiability Checking

We now introduce a satisfiability checking algorithm. This satisfiability checking is done before the invocation of a component service. Suppose that, before the invocation of a component service S_i, CS is at the active state s_a. The satisfiability of the rLTC at s_a will be checked before S_i is invoked. If it is satisfiable, then it will invoke S_i as usual. Otherwise, some mitigation procedures will be triggered. A mitigation procedure could be to invoke a faster backup service S'_i instead of S_i.

Given a composite service CS with n component services $E = \{S_1, S_2, \ldots, S_n\}$, let us explain how the satisfiability checking is performed prior to the invocation of a component service $S_i \in E$. Let $\{t_1, t_2, \ldots, t_n\}$ and $\{st_1, st_2, \ldots, st_n\}$, where $t_i \in \mathcal{L}_U$ and $st_i \in \mathbb{R}_{\geq 0}$, be the set of parametric response times and stipulated response times for component services in E respectively. We denote by $T_{CS} = \{(t_1, st_1), \ldots, (t_n, st_n)\}$ the stipulated response time information of component services CS.

Algorithm 4: checkSat(LTS_{CS}, s_a, r, T_{CS})

input : The LTS of the parametric composite service CS
input : The active state $s_a \in Q$
input : The elapsed time $r \in \mathbb{R}_{\geq 0}$
input : The stipulated response time information $T_{CS} = \{(t_1, st_1), \ldots, (t_n, st_n)\}$
output: True if the local time constraint at s_a is satisfiable, false otherwise

1 **return** $Is_Sat((\bigwedge_{1 \leq i \leq n} t_i \leq st_i) \Rightarrow (s_a.rLTC \wedge (r_f = r)))$;

We give in Algorithm 4 the algorithm checking the satisfiability of rLTC at state $s_a \in Q$. With the assumption that all component services will reply within their stipulated response times ($\bigwedge_{1 \leq i \leq n} t_i \leq st_i$), checkSat checks whether the rLTC at state s_a can be satisfied with free variables r_f substituted with the actual elapsed time $r \in \mathbb{R}_{\geq 0}$. The function Is_Sat at line 1 will return true if the input constraint is satisfiable.

6.5. Termination and Soundness

We now prove the termination and soundness of the synthesis of synthRLTC.

6.5.1. Termination

PROPOSITION 6.1. *Let CS be a service model, s be a state in LTS_{CS}. Then* synthRLTC(CS, LTS_{CS}, s) *terminates.*

PROOF. *Observe that Algorithm 3 is recursive (on line 9). However, due to the acyclic nature of LTS_{CS} (from Lemma 5.1) and the fact that Algorithm 3 is called recursively on the successors of the current state, then no state is explored more than once. This ensures termination.* □

6.5.2. Soundness. Theorem 6.3 will formally state the correctness of our runtime refinement algorithm. It generalizes Theorem 5.11 to the case of runtime refinement. We first need the following lemma. The following lemma states that, given a run ρ of LTS_{CS}, there exists a unique equivalent run in $LTS_{CS[\pi]}$, provided π satisfies the parametric constraint associated with the last state of ρ. The result is dual to what we proved in Lemma 5.4.

LEMMA 6.2. *Let CS be a parametric service model, and let π be a parameter valuation. Let ρ be a run of LTS_{CS} ending in a state (v_n, P_n, C_n, D_n).*
For any $\pi \models C_n{\downarrow_U}$, there exists a unique run of $LTS_{CS[\pi]}$ equivalent to ρ.

PROOF. By induction on the length of the runs. We prove in fact a slightly stronger result: given a state (v', P', C', D') of a run ρ' in LTS_{CS}, and given a state (v, P, C, D) of the equivalent run ρ in $LTS_{CS[\pi]}$, we show that these states are not only equivalent, but also that $C = C'[\pi]$.

Base case. From Definition 4.4, the initial state of LTS_{CS} is $(v_0, P_0, C_0, 0)$. The initial state of $LTS_{CS[\pi]}$ is $(v_0, P_0[\pi], C_0[\pi], 0)$. Since $C_0 = \mathit{true}$ then $C_0 = C_0[\pi]$. Hence the result trivially holds in that case.
Induction step. Assume ρ is a run of LTS_{CS} of length m reaching state (v'_1, P'_1, C'_1, D'_1). Let (v'_2, P'_2, C'_2, D'_2) be the successor state of (v'_1, P'_1, C'_1, D'_1) via action a in ρ. Let $\pi \models C'_2{\downarrow_U}$. Assume there exists a unique run of $LTS_{CS[\pi]}$ equivalent to ρ and of length m, reaching state (v_1, P_1, C_1, D_1). From Definition 5.3, it holds that $v_1 = v'_1$ and $P_1 = P'_1[\pi]$. From the induction hypothesis, it holds that $C_1 = C'_1[\pi]$.
Assume (v'_2, P'_2, C'_2, D'_2) is obtained from (v'_1, P'_1, C'_1, D'_1) by applying rule *rSInv* in Fig. 6. Recall that $C_1 = C'_1[\pi]$; since $P_1 = P'_1[\pi]$ (from Definition 5.3), we can apply rule *rSInv* to (v_1, P_1, C_1, D_1), yielding a state (v_2, P_2, C_2, D_2). From Fig. 6, we know that $C_2 = (x = t_S \wedge (C_1^{\uparrow}))$ and $C'_2 = (x = t_S \wedge (C_1'^{\uparrow}))$. Now, we have:

$$\begin{aligned}
C_2 &= \big(x = t_S \wedge (C_1^{\uparrow})\big) \\
&= \big(x = t_S \wedge (C'_1[\pi])^{\uparrow}\big) && \text{(induction hypothesis)} \\
&= \big(x = t_S \wedge (C'_1 \wedge \textstyle\bigwedge_{u_i \in U} u_i = \pi_i)^{\uparrow}\big) && \text{(definition of valuation)} \\
&= \big(x = t_S \wedge (C'_1)^{\uparrow}\big) \wedge \textstyle\bigwedge_{u_i \in U} u_i = \pi_i && \text{(property of time elapsing)} \\
&= C'_2 \wedge \textstyle\bigwedge_{u_i \in U} u_i = \pi_i && \text{(definition of } C'_2\text{)} \\
&= C'_2[\pi] && \text{(definition of valuation)}
\end{aligned}$$

Note that adding $x = t_S$ while keeping satisfiability of the expression is only true because $\pi \models C'_2{\downarrow_U}$. This implies that $C'_2[\pi]$ is non-empty, hence the state (v_2, P_2, C_2, D_2) is a valid state. In addition, since $P_1 = P'_1[\pi]$ and rule *rSInv* derives to *Stop*, then $P_2 = P'_2[\pi]$. Similarly, variables are updated in the same manner on both sides, hence $v_2 = v'_2$. The proof is similar for other rules in Fig. 6.
The proof of uniqueness is identical to that of Lemma 5.4.

□

THEOREM 6.3. *Let CS be a service model with stipulated response time information T_{CS}. Let LTS_{CS} be the LTS of CS. Let s be the current state in LTS_{CS} and r be the current elapsed time.*
Fix $\pi \models \mathsf{synthRLTC}(LTS_{CS}, s, r, T_{CS})$. Then:

(1) *there exists a non-empty state s_π in $LTS_{CS[\pi]}$ equivalent to s;*
(2) *any complete run of the sub-LTS of $LTS_{CS[\pi]}$ starting from s_π ends in a good state;*
(3) *for all good states (v, P_g, C, d) in the sub-LTS of $LTS_{CS[\pi]}$ starting from s_π, then $d \leq T_G$.*

PROOF.

(1) From Lemma 6.2.
(2) From Definition 4.2, the sub-LTS of $LTS_{CS[\pi]}$ starting from s_π contains the successors of s_π in $LTS_{CS[\pi]}$, and hence any complete run of the sub-LTS of $LTS_{CS[\pi]}$ starting from s_π corresponds to the end of some complete run of $LTS_{CS[\pi]}$. From

Lemma 5.9, any complete run of $LTS_{CS[\pi]}$ ends in a good state, which gives the result.

(3) Any good state of the sub-LTS of $LTS_{CS[\pi]}$ starting from s_π is also a good state of $LTS_{CS[\pi]}$. From Lemma 5.10, for all good state of $LTS_{CS[\pi]}$, $d \leq T_G$, which gives the result.

□

6.6. Discussion

Termination. From Theorem 6.1, our method terminates thanks to the fact that BPEL composite services do not support recursion, and thanks to Assumption 1 on the loop activities ensuring that the upper bound on the number of iterations and the time of execution is known. We discuss how to enforce this assumption in the presence of loops in the composite service. The upper bound on the number of iterations could be either inferred by using loop bound analysis tool (*e.g.*, [Ermedahl et al. 2007]), or could be provided by the user otherwise. In the worst case, an option is also to set up a bound arbitrary but "large enough". Concerning the maximum time of loop executions, it could be enforced by using proper timeout mechanism in BPEL.

Time for internal operations. For simplicity, we do not account for the time taken for the internal operations of the system. In reality, the time taken by the internal operations may become significant, especially when the process is large. We can provide a more accurate synthesis of the constraints by including an additional constraint $t_{overhead} \leq b$, where $t_{overhead} \in \mathbb{R}_{\geq 0}$ is a time overhead for an internal operation, and $b \in \mathbb{R}_{\geq 0}$ is a machine dependent upper bound for $t_{overhead}$. The method to obtain an estimation of b is beyond the scope of this work; interested readers may refer to, *e.g.*, [Moser et al. 2008].

Completeness of synthRLTC*.* The rLTC computed by our algorithm synthRLTC is still incomplete in general, with the same reason for the incompleteness of the sLTC computed by synthSLTC as discussed in Section 5.7. Nevertheless, it helps to mitigate the problem of incompleteness of the sLTC with LTS simplification, as illustrated in Section 6.1.

Bad activities. The bad activities are the activities triggered when timeout occurs. For the running example SMIS, it is a reply activity that reports the user on the timeout of a composite service. As an additional example, it could also be an invocation activity to log the timeout event upon the timeout of a composite service. With the rule of thumb that a bad activity is always triggered upon the timeout of a composite service, identifying a bad activity would become an easy task; devising techniques for (semi-)automating this task is left as future work. On the other hand, specifying bad activities is not mandatory. If the user cannot identify a bad activity in the composite service, (s)he has the option not to specify any. Doing so, all activities in the composite service are treated as good activities. This implies that the synthesized constraints only provide the following guarantee: any possible complete run of the composite service is able to satisfy the global time requirement upon completion. It does not consider the situation where the execution of a complete run could directly lead to the violation of the global time requirement, *e.g.*, the complete run that contains s_0, s_2, and s_4 in Fig. 10.

7. EVALUATION

In this section, we apply our method to several case studies. First, we briefly present our implementation (Section 7.1). Then, we describe the case studies we use (Sec-

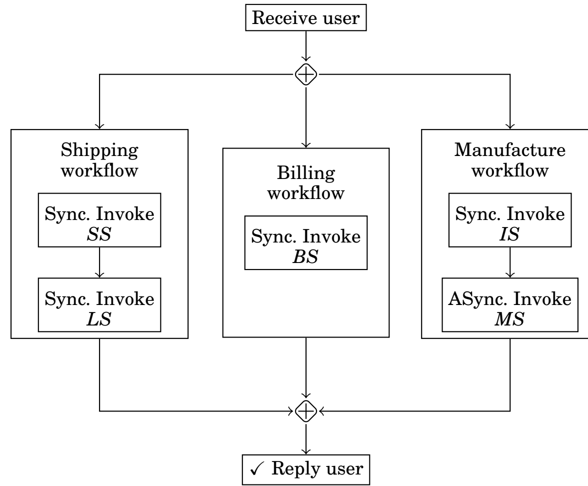


Fig. 16: Computer Purchasing Service (CPS)

tion 7.2). Then, we evaluate our method for the synthesis of local time requirement at the design time (Section 7.3). And last, we focus on the evaluation of the runtime adaptation of a composite service (Section 7.4).

7.1. Implementation

We have implemented our algorithms for synthesizing sLTC and rLTC (*viz.*, synthSLTC and synthRLTC) in SELAMAT, a tool developed in C♯. The tool and case studies that will be described in Section 7.2 can be downloaded at [Tan et al. 2015]. The simplification of the final results of sLTC and rLTC is achieved using Microsoft Z3 [de Moura and Bjørner 2008]. For the runtime adaptation, we make use of Apache ODE 1.3.6 as runtime engine module (*RE*). The service monitoring module (*SM*) is developed in C♯, which uses Microsoft Z3 for the satisfiability checking.

7.2. Case Studies

Stock Market Indices Service (SMIS). This is the running example introduced in Section 3.

Computer Purchasing Services (CPS). The goal of a CPS (such as Dell.com) is to allow a user to purchase a computer system online using credit cards. Our CPS makes use of five component services, namely Shipping Service (*SS*), Logistic Service (*LS*), Inventory Service (*IS*), Manufacture Service (*MS*), and Billing Service (*BS*). The global time requirement of the CPS is to respond within three seconds. The CPS workflow is shown in Fig. 16. The CPS starts upon receiving the purchase request from the client with credit card information, and the CPS spawns three workflows (*viz.*, shipping workflow, billing workflow, and manufacture workflow) concurrently. In the shipping workflow, the shipping service provider is invoked synchronously for the shipping service on computer systems. Upon receiving the reply, *LS* (which is a service provided by the internal logistic department) is invoked synchronously to record the shipping schedule. In the billing workflow, the billing service (which is offered by a third party merchant) is invoked synchronously for billing the customer with credit card information. In the manufacture workflow, *IS* is invoked synchronously to check for the availability of the goods. Subsequently, *MS* is invoked asynchronously to update the manufacture department regarding the current inventory stock. Upon receiving the

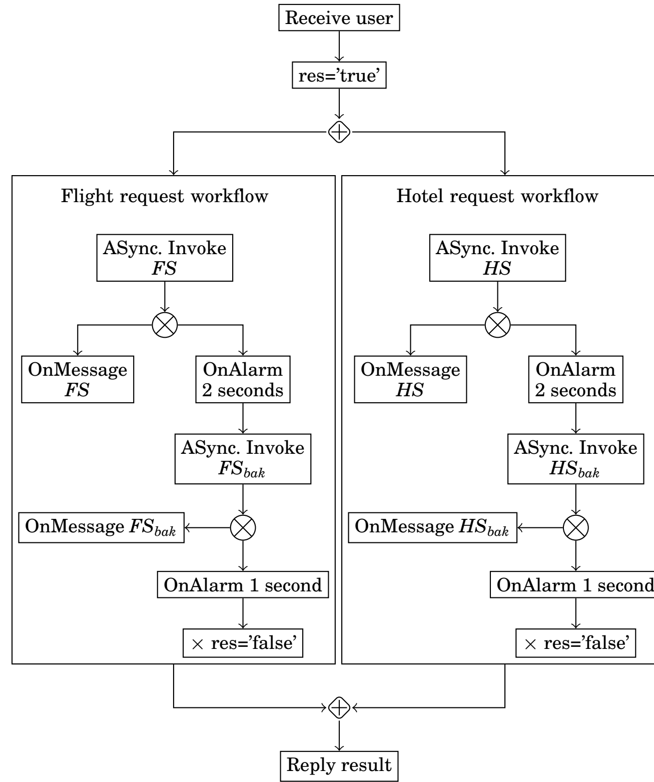


Fig. 17: Travel Booking Service (TBS)

reply message from *LS* and *BS*, the result of the computer purchasing will be returned to the user.

Travel Booking Service (TBS). The goal of a travel booking service (TBS) (such as Booking.com) is to provide a combined flight and hotel booking service by integrating two independent existing services. TBS provides an SLA for its subscribed users, saying that it must respond within five seconds upon request. The travel booking system has four component services, namely Flight Service (*FS*), Backup Flight Service (FS_{bak}), Hotel Service (*HS*) and Backup Hotel Service (HS_{bak}). The TBS workflow is given in Fig. 17. Upon receiving the request from users, the variable *res* is assigned to true. After that, TBS spawns two workflows (*viz.*, a flight request workflow, and a hotel request workflow) concurrently. In the flight request workflow, it starts by invoking *FS*, which is a service provided by a flight service booking agent. If service *FS* does not respond within two seconds, then *FS* is abandoned, and another backup flight service FS_{bak} is invoked. If FS_{bak} returns within one second, then the workflow is completed; otherwise the variable *res* is assigned to false. The hotel request workflow shares the same process as the flight request workflow, by replacing *FS* with *HS* and FS_{bak} with HS_{bak}. The booking result will be replied to the user if *res* is true; otherwise, the user will be informed of the booking failure.

Rescue Team Service (RS). The goal of a Rescue Team service (RS) is to identify the place, weather, and nearest rescue team, by the longitude and latitude on Earth. RS makes use of three component services, namely Terra Service (*TS*), Weather Ser-

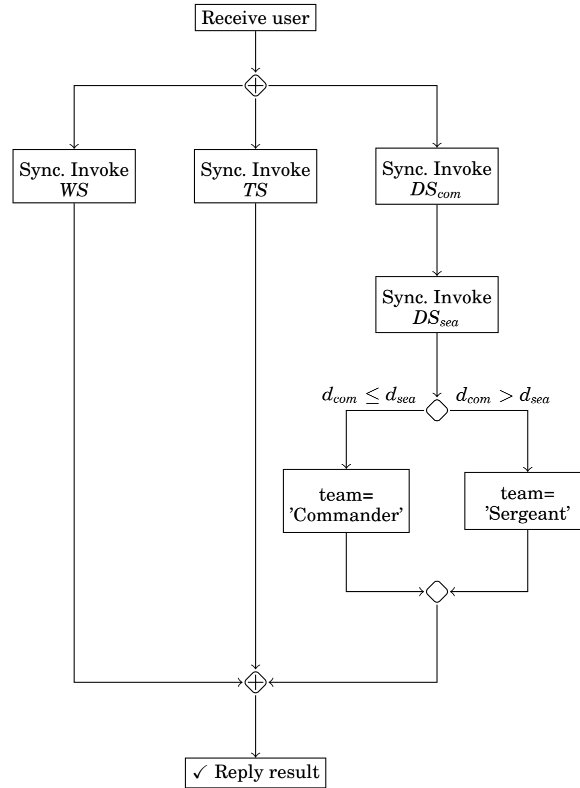


Fig. 18: Rescue Team Service (RS)

vice (*WS*) and Distance Service (*DS*). The global requirement of the RS is to respond within five seconds. The RS workflow is given in Fig. 18. RS starts upon receiving longitude and latitude coordinates from the user. After that, it invokes Terra Service (*TS*), Weather Service (*WS*), and Distance Service (*DS*) concurrently. Service *TS* (resp. *WS*) will return the name of the place (resp. the weather information) that corresponds to the longitude and latitude. *DS* is used to calculate the distance between each rescue team and the event location. In particular, DS_{com} and DS_{sea} are used to calculate the distance between commander team and sergeant team to the event location. If the distance to the event of the commander team (d_{com}) is not larger than the distance to the event of the sergeant team (d_{sea}), then the commander team will be chosen. Otherwise, the sergeant team will be chosen. Subsequently, the place, weather and rescue team information is returned to the user.

7.3. Synthesis of Local Time Requirement

7.3.1. Environment of the Experiments. We synthesize the sLTC and rLTC for four case studies on a system using Intel Core I5 2410M CPU with 4 GiB RAM.

7.3.2. Evaluation Results. The details of the synthesis are shown in Table I. The **#states** and **#transitions** columns provide the information of number of states and transitions of the LTS, respectively. We repeated all experiments 30 times; we report here the average time for each experiment. The **sLTC** and **rLTC** columns provide the average time (in seconds) spent for synthesizing sLTC (for the entire LTS), and rLTC

Case Studies	#states	#transitions	sLTC (s)	rLTC (s)
SMIS	14	13	0.0076	0.0078
TBS	683	3677	1.8501	1.9000
CPS	120	119	0.0529	0.0559
RS	85	134	0.0701	0.0733

Table I: Synthesis for sLTC and rLTC

Case Studies	#states	#transitions	sLTC (s)
SMIS	17	16	0.0090
TBS	938	4614	1.9811
CPS	144	143	0.0626
RS	117	166	0.0740

Table II: Synthesis for sLTC based on [Tan et al. 2013]

(for each state in the LTS), respectively. TBS takes a longer time than SMIS, CPS, and RS for synthesizing sLTC and rLTC, as it contains a larger number of states and transitions compared to SMIS, CPS, and RS. Nevertheless, since both sLTC and rLTC are synthesized offline, the time for synthesizing the constraints (around one second) for TBS is considered to be reasonable.

The main overhead on synthesizing sLTC and rLTC is due to the calculation of the constraint component C of each new state $s = (v, P, C, D)$. Calculation of a constraint component C' is required for each transition in the LTS, in order to create the successor state $s' = (v', P', C', D')$. If two created states s, s' in the LTS are found to be equal, they will be merged into a single state. Therefore, the number of transitions will have a greater effect than the number of states on the time spent for synthesizing sLTC and rLTC.

For comparison with [Tan et al. 2013], we also provided the information of synthesis of sLTC based on [Tan et al. 2013] in Table II (sLTC is given in seconds). The additional “and/or states” (used in [Tan et al. 2013] for synthesizing the sLTC) yield an increased number of both states and transitions. As a consequence, this yields an increased time for synthesizing the sLTC, for all case studies.

The synthesized sLTC for SMIS is shown in Fig. 13, while the synthesized sLTC for CPS, TBS, and RS are shown in Fig. 19. All the sLTC are simplified and in DNF form. It is worth noting that the sLTC of CPS and RS can be represented in one line representation (*i. e.*, only one inequality) after simplification. Note that t_{MS} does not appear in the sLTC of CPS. The reason is that *MS* is invoked asynchronously without expecting a response; therefore its response time is irrelevant to the global time requirement of CPS.

The synthesized rLTC are used for runtime adaptation during runtime. We evaluate the runtime adaptation of a composite service with rLTC in the following section.

7.4. Runtime Adaptation

We now conduct experiment to evaluate the *overhead* of the runtime adaptation and the *improvement* caused by the runtime adaptation on the conformance of the global time requirement. Specifically, we attempt to answer the following questions.

Q1. What is the *overhead* of the runtime adaptation?

Q2. What is the *improvement* provided by the runtime adaptation?

$(t_{SS} + t_{LS} + t_{IS} + t_{BS}) \leq 3$

(a) sLTC for CPS

$(t_{TS} + t_{WS} + 2 \cdot t_{DS}) \leq 5$

(b) sLTC for RS

$((2 \cdot t_{HSbak} < t_{FSbak}) \wedge (2 \cdot t_{FSbak} < t_{HSbak}) \wedge (t_{HSbak} < 1) \wedge (t_{FSbak} < 1))$
$\vee ((t_{HSbak} < 1) \wedge (t_{FSbak} < 1) \wedge (t_{FSbak} + t_{HSbak} \leq 1))$
$\vee ((t_{HSbak} < 1) \wedge (t_{FS} < 2)) \vee ((t_{HS} < 2) \wedge (t_{FSbak} < 1)) \vee ((t_{HS} < 2) \wedge (t_{FS} < 2))$

(c) sLTC for TBS

Fig. 19: Synthesized sLTC

Case Studies	Avg. #SAT	Avg. SAT runtime (s)
SMIS	1.74	13
TBS	2.25	17
CPS	4.00	27
RS	4.00	19

Table III: Satisfiability checking

7.4.1. Environment of the Experiments. The evaluation was conducted using two different physical machines, connected by a 100 Mbit LAN. One machine is running ApacheODE [Foundation 2007] to host the *RE* module to execute the BPEL program, configured with Intel Core I5 2410M CPU with 4GiB RAM. The other machine hosts the *SM* module, configured with Intel I7 3520M CPU with 8GiB RAM.

To test the composite service under controlled situation, we introduce the notion of *execution configuration*. An execution configuration defines a particular execution scenario for the composite service. Formally, an execution configuration E is a tuple (M, R), where M decides which path to choose for an `<if>` activity and R is a function that maps a component service S_i to a real value $r \in \mathbb{R}_{\geq 0}$, which represents the response time of service s_i. We discuss how an execution configuration $E = (M, R)$ is generated. M is generated by choosing one of the branches of an `<if>` activity uniformly among all possible branches.

Let CS be a composite service, where a component service S_i of CS has a stipulated response time $v_i \in \mathbb{R}_{\geq 0}$. Then $R(S_i)$ will be assigned with a response time within the stipulated response time v_i with a probability of $p_c \in \mathbb{R}_{\geq 0} \cap [0, 1]$. p_c is the *response time conformance threshold*. More specifically, $R(S_i)$ will be assigned with a value in $[0, v_i]$ uniformly with a probability of p_c, and assigned to a value in $(v_i, v_i + t_e]$ uniformly with a probability of $1 - p_c$. $t_e \in \mathbb{R}_{\geq 0}$ is the *exceeding threshold*; and assume after $v_i + t_e$ seconds, the component service S_i will be automatically timeout by RE to prevent an infinite delay.

Given a composite service CS, and an execution configuration E, a *run* is denoted by $r(CS, AM, E)$, where the first argument is the composite service CS that is running, the second argument $AM \in \{rr, \varnothing\}$ is the adaptive mechanism where rr denotes the runtime adaptation, and $\varnothing$ denotes no runtime adaptation. Two runs $r(CS, AM, E)$ and $r(CS', AM', E')$ are equal if $CS = CS'$, $AM = AM'$ and $E = E'$. Note that all equal runs have the same execution paths and response times for all service invocations.

We use two kinds of instrumentation for runtime engines used for adaptive and non-adaptive runs, respectively. For both adaptive and non-adaptive runs, we instrument both runtime engines to execute conditional statements based on M. For adaptive runs only, we instrument the runtime engine according to Section 6.2.

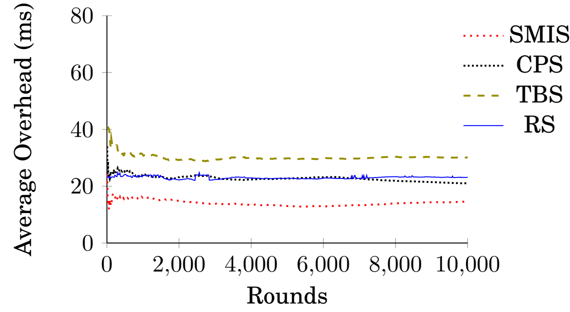


Fig. 20: Overhead of runtime monitoring

	p_c	N_{se}	N_e	**Improvement (%)**	**Avg. Backup Service**
SMIS	0.9	9441	8976	5.18	0.127
	0.8	9211	8374	10.00	0.352
	0.7	8109	6965	16.42	0.577
	0.6	7593	6348	19.61	0.702
TBS	0.9	10000	9743	2.64	0.384
	0.8	10000	9364	6.79	0.779
	0.7	10000	8460	18.20	0.948
	0.6	10000	7700	29.87	1.05
CPS	0.9	9523	8809	8.11	1.259
	0.8	9241	7156	29.14	1.509
	0.7	8504	6108	39.23	2.014
	0.6	8430	5650	49.20	2.578
RS	0.9	8181	7271	12.52	1.787
	0.8	7201	7011	2.71	1.589
	0.7	6590	5227	26.08	1.659
	0.6	5609	4146	35.29	1.54

Table IV: Improvement of runtime conformance

7.4.2. Evaluation Results. We conducted two experiments Exp1 and Exp2, to answer the research questions Q1 and Q2, respectively. Each experiment goes through 10,000 rounds of simulation, and an execution configuration E is generated for each round of simulation. Given a composite service CS, we assume that for each component service S_i with a stipulated response time v_i, there exists a backup service S_i', with a stipulated response time $v_i/2$ and a conformance threshold of 1. Suppose that before the invocation of a component service S_i, CS is at active state s_a. The satisfiability of the rLTC at s_a will be checked using Algorithm 4 before S_i is invoked. If it is satisfiable, then it will invoke S_i as usual. Otherwise, as an mitigation procedure, the faster backup service S_i' will be invoked instead.

We now describe both experiments Exp1 and Exp2.

Experiment Exp1. Given a composite service CS, in order to measure the overhead, we use an execution configuration $E = (M, Q)$ for an adaptive run $r(CS, rr, E)$, and non-adaptive run $r(CS, \varnothing, E)$. We have modified the runtime adaptation mechanism for rr

such that, if the rLTC of the active state is checked to be unsatisfiable, component service S_i will still be used (instead of S'_i). The purpose for this modification is to make $r(CS, rr, E)$ and $r(CS, \varnothing, E)$ invoke the same set of component services, so that we can effectively compare the overhead of $r(CS, rr, E)$.

Results. Suppose at round k, the times spent for $r(CS, rr, E)$ and $r(CS, \varnothing, E)$ are $r^k_{rr} \in \mathbb{R}_{\geq 0}$ time units and $r^k_{\varnothing} \in \mathbb{R}_{\geq 0}$ time units respectively. The overhead O_k at round k is the time difference between r^k_{rr} and $r^k_{\varnothing}$, *i.e.*, $O_k = r^k_{rr} - r^k_{\varnothing}$. The average overhead at round k is calculated using Equation 1.

$$Average\ overhead = (\sum_{i=1}^{k} O_i)/k \tag{1}$$

The main source of overhead for runtime adaptation comes from the satisfiability checking with Algorithm 4. We make use of the state-of-the-art SMT solver Z3 [de Moura and Bjørner 2008] for this purpose. Other sources of overhead include update of active state in SM, and communications between SM and RE.

The experiment results can be found in Fig. 20. The average overheads of SMIS, CPS, TBS, and RS after 10,000 rounds are 15 ms, 21 ms, 30 ms, and 23 ms respectively. The results convey to us that the additional operations involved in the runtime adaptation, including the satisfiability checking, can be done efficiently.

We further evaluate the overhead on satisfiability checking. Table III shows the results of satisfiability checking. The average number of satisfiability checking for each round (Avg. #SAT) is calculated using Equation 2 where N_i is the total number of satisfiability checking for i-th round and r is the total number of running rounds. The average time (given in milliseconds) spent on satisfiability checking for each round (Avg. SAT runtime) is calculated using Equation 3, where T_i is the time spent on satisfiability checking for i-th round. Table III shows that the satisfiability checking has contributed most of the overhead of runtime adaptation.

$$Average\ \#SAT = (\sum_{i=1}^{r} N_i)/r \tag{2}$$

$$Avg.\ SAT\ runtime = (\sum_{i=1}^{r} T_i)/r \tag{3}$$

Experiment Exp2. In this second experiment, we measure the improvement for the conformance of global constraints due to rr. Given a composite service CS, an execution configuration E, two runs $r(CS, rr, E)$ and $r(CS, \varnothing, E)$ are conducted for each round of simulation. N_{se} is the number of executions that satisfy global constraints for composite service with rr, and N_e is the number of executions that satisfy global constraints for composite service without rr, the improvement is calculated by Equation 4.

$$Improvement = \frac{(N_{se} - N_e) * 100}{N_e} \tag{4}$$

Results. The experiment results can be found in Table IV. The *Improvement* (%) column provides the information of improvement (in percentage) that is calculated using Equation 4. The *Avg. Backup Service* column provides the average number of backup service used (calculated by summing the number of backup services used for 10,000 rounds, and divided by 10,000).

The decrement of p_c represents the undesired situation where component services have a higher chance for not conforming to their stipulated response time. This may be due to situations such as poor network conditions. For each example, the improvement provided by the runtime adaptation increases when p_c decreases. This shows that runtime adaptation improves the conformance of global time requirement. In addition, the average number of backup service used increases when p_c decreases. This shows the adaptive nature of runtime adaptation with respect to different p_c – more corrective actions are likely to perform when the chances that component services do not satisfy their stipulated response time increase.

Answer to Research Questions. The results in Exp1 and Exp2 have shown that the runtime adaptation has a low overhead, and improves the runtime conformance, especially when the response time conformance threshold of the component services is low.

7.5. Threats to Validity

There are several threats to validity. The first threat to validity is due to the fact that we assume a uniform distribution of response time for evaluation of runtime adaptation. To address this issue, more experimentations with real-world services should be performed. This said, our experiments on case studies provide a first idea that our assumptions are realistic.

The second threat to validity is stemmed from our choice to use a few example values as experimental parameters, that include global constraints and termination thresholds, in order to cope with the combinatorial explosion of options. To resolve this problem, it is clear that even more experimentations with different case studies and experimental parameters should be performed, so that we could further investigate the effects that have not been made obvious by our case studies and experimental parameters.

8. RELATED WORK

Constraint synthesis for scheduling problems. Our work shares common techniques with work for constraint synthesis for scheduling problems. The use of models such as parametric timed automata (PTAs) [Alur et al. 1993] and parametric time Petri nets (PTPNs) [Traonouez et al. 2009] for solving such problems has received recent attention. In particular, in [Cimatti et al. 2008; Le et al. 2010; Fribourg et al. 2012], parametric constraints are inferred, guaranteeing the feasibility of a schedule using PTAs extended with stopwatches (see, *e.g.*, [Adbeddaïm and Maler 2002]). In [André et al. 2014], we proposed a parametric, timed extension of CSP, to which we extended the "inverse method", a parameter synthesis algorithms preserving the discrete behavior of the system (see, *e.g.*, [André and Soulat 2013]). Although PTAs or PTPNs might have been used to encode (part of) the BPEL language, our work is specifically adapted and optimized for synthesizing local timing constraint in the area of service composition.

Analysis with LTSs in Web services. Our method is related to using LTSs for analysis purpose in Web services. In [Bianculli et al. 2011], the authors propose an approach to obtain behavioral interfaces in the form of LTSs of external services by decomposing the global interface specification. It also has been used in model checking the safety and liveness properties of BPEL services. For example, Foster *et al.* [Foster 2006; Foster et al. 2006] transform BPEL process into FSP [Magee and Kramer 2006], subsequently using a tool named "WS-Engineer" for checking safety and liveness properties. Simmonds *et al.* [Simmonds et al. 2010] propose a user-guided recovery framework for

Web services based on LTSs. Our work uses LTSs in synthesizing local time requirement.

Finding suitable quality of service for the system. Our method is related to the finding of a suitable quality of service (QoS) for the system [Yu et al. 2007]. The authors of [Yu et al. 2007] propose two models for the QoS-based service composition problem: a combinatorial model and a graph model. The combinatorial model defines the problem as a multidimension multichoice 0-1 knapsack problem. The graph model defines the problem as a multiconstraint optimal path problem. A heuristic algorithm is proposed for each model: the WS-HEU algorithm for the combinatorial model and the MCSP-K algorithm for the graph model. The authors of [Ardagna and Pernici 2005] model the service composition problem as a mixed integer linear problem where constraints of global and local component service can be specified. The difference with our work is that, in their work, the local constraint is specified, whereas in ours, the local constraint is synthesized. An approach of decomposing the global QoS to local QoS has been proposed in [Alrifai and Risse 2009]. It uses the mixed integer programming (MIP) to find optimal decomposition of QoS constraint. However, the approach only concerns simple sequential composition of Web services method call, without considering complex control flows and timing requirements.

Response time estimation. Our approach is also related to response time estimation. In [Kraft et al. 2009], the authors propose to use linear regression method and a maximum likelihood technique for estimating the service demands of requests based on their response times. [Menascé 2004] has also discussed the impact of slow services on the overall response time on a transaction that use several services concurrently. Our work is focused on decomposing the global requirement into local requirement, which is orthogonal to these works. Our recent work [Li et al. 2014] complements with this work by proposing a method on building LTCs that under-approximate the sLTC of a composite service. The under-approximated LTCs consisting of independent constraints over components, which can be used to improve the design, monitoring and repair of component-based systems under time requirements.

Service monitoring. Our method is related to service monitoring. Moser *et al*. [Moser et al. 2008] present VieDAME, a non-intrusive approach to monitoring. VieDAME allows monitoring of BPEL composite service on quality of service attributes, and existing component services are replaced based on different replacement strategies. They make use of the aspect-oriented approach (AOP); therefore the VieDAME engine adapter could be interwoven into the BPEL runtime engine at runtime. Baresi *et al.* [Baresi and Guinea 2011] propose an idea of self-supervising BPEL processes by supporting both service monitoring and recovery for BPEL processes. They propose the use of Web Service Constraint Language (WSCoL) to specify the monitoring directives to indicate properties need to be hold during the runtime of composite service. They also make use of the AOP approach to integrate their monitoring adapters with the BPEL runtime engine. Our work is orthogonal to the aforementioned works, as we do not assume any particular service monitoring framework for monitoring the composite service, and those methods can be used to aid the monitoring approach, as discussed in Section 6.2. Our previous work [Tan et al. 2014] proposes an automated approach based on a genetic algorithm to calculate the recovery plan that can guarantee the satisfaction of functional properties of the composite service after recovery.

Verification of services. Concerning verification of services, Filieri *et al*. [Filieri et al. 2011] focus on checking the reliability of component (service)-based systems. They make use of Discrete Time Markov Chain (DTMC) to check the reliability of models at runtime. Our previous works [Chen et al. 2013; Chen et al. 2014] develop a tool to

verify combined functional and non-functional requirements of Web service composition. In contrast, the current work focuses on response time: given the global response time of the composite service, we synthesize the response time requirement for component services at design time and refine it at runtime. Schmieders *et al*. [Schmieders and Metzger 2011] proposed the SPADE approach. SPADE invokes the BOGOR model checker to model check the SLAs at design time and at runtime. Our work is different from theirs in two aspects. First, we focus on the synthesis of the local time requirement, which is a formal requirement on the response time requirement of component services. Second, during runtime, [Schmieders and Metzger 2011] performs model checking on a given state to check whether an adaptation is needed. In contrast, we have precomputed the constraints for every state at design time. Therefore, we only require evaluation of constraints by substituting the free variables during runtime, and this allows a more efficient runtime-analysis.

9. CONCLUSION AND FUTURE WORKS

Conclusion. We have presented a novel technique for synthesizing local time constraints for the component services of a composite service *CS*, knowing its global time requirement. Our approach is based on the analysis of the LTS of a composite service by making use of parameterized timed techniques. The local time constraint can guarantee the the satisfaction of the global response time requirement. Our proposed techniques consist of static and dynamic checking of global time requirement based on the sLTC and rLTC of component services respectively.

During design time of a composite service, we propose a synthesis algorithm, that utilizes the parametric constraints from the LTS, to synthesize static local time constraint (sLTC) for component services. The sLTC is then used to select a set of component services that could collectively satisfy the global time requirement in design time.

Then, during the runtime of a composite service, we propose the usage of the runtime information to weaken the sLTC, which becomes the refined local time constraint (rLTC). In particular, two pieces of runtime information have been leveraged – the execution path that has been taken by the composite service, and the elapsed time of the composite service. The rLTC is then used to validate whether the composite service can still satisfy the global time requirement at runtime. We have implemented the approach into a tool SELAMAT, and applied it to four case studies.

Our experiments show that the computation time is always smaller than our previous approach [Tan et al. 2013], and that the runtime refinement leads to an improvement of the global time requirement, while limiting the overhead.

Future works. We plan to further improve and develop the technique presented in this paper.

First, we would like to benefit from optimizations techniques developed for other formalisms such as timed automata and parametric timed automata, such as convex state merging [André et al. 2013], and adapt them to our setting.

Second, we will investigate the usage of soft deadlines that allow to run a service with a delay, possibly with an acceptable penalty.

Our work so far deals with exact response times. A different approach would be to consider that the response time should be fulfilled with some probability. In that setting, the goal would be to synthesize the values for the timing parameters such that the response time is indeed below the threshold with a given probability. To achieve this, we could reuse recent works involving probabilities and timing parameters (*e.g.*, [Jovanović and Kwiatkowska 2014; Ceska et al. 2014]). Even more challenging would

be to combine both kinds of parameters (timing parameters and probabilistic parameters), so as to infer the probability under which the response time can be fulfilled.

The construction of the LTS of a BPEL process is so far achieved in our implementation on a monocore computer (or in fact on a multicore computer but using a single core). Extending this construction to multicore computers while performing parameter synthesis would be challenging. A parallel algorithm for Büchi emptiness checking in timed automata, proposed in [Laarman et al. 2013], could be extended to BPEL and to the parametric case.

Finally, we could extend our current approach to other domains that share similar problems, for example wireless sensor networks [Pottie and Kaiser 2000].

Acknowledgement

Étienne André, Jin Song Dong and Yang Liu are partially supported by CNRS STIC-Asie project CATS ("Compositional Analysis of Timed Systems"). Étienne André is partially supported by the ANR national research program ANR-14-CE28-0002 PACS ("Parametric Analyses of Concurrent Systems").

REFERENCES

Yasmina Adbeddaïm and Oded Maler. 2002. Preemptive Job-Shop Scheduling using Stopwatch Automata. In *TACAS (Lecture Notes in Computer Science)*, Vol. 2280. Springer-Verlag, 113–126.

Mohammad Alrifai and Thomas Risse. 2009. Combining global optimization with local selection for efficient QoS-aware service composition.. In *WWW*. ACM, 881–890.

Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theoretical computer science* 126, 2 (1994), 183–235. DOI:http://dx.doi.org/10.1016/0304-3975(94)90010-8

Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. 1993. Parametric real-time reasoning. In *STOC*. ACM, 592–601.

Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, IBM, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. 2007. *Web Services Business Process Execution Language Version, version 2.0*.

Étienne André. 2013. Dynamic Clock Elimination in Parametric Timed Automata. In *FSFMA (OpenAccess Series in Informatics (OASIcs))*, Vol. 31. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, 18–31.

Étienne André, Laurent Fribourg, and Romain Soulat. 2013. Merge and Conquer: State Merging in Parametric Timed Automata. In *ATVA (Lecture Notes in Computer Science)*, Vol. 8172. Springer, 381–396.

Étienne André, Yang Liu, Jun Sun, and Jin Song Dong. 2014. Parameter Synthesis for Hierarchical Concurrent Real-Time Systems. *Real-Time Systems* 50, 5-6 (sep 2014), 620–679. DOI:http://dx.doi.org/10.1007/s11241-014-9208-6

Étienne André and Romain Soulat. 2013. *The Inverse Method*. ISTE Ltd and John Wiley & Sons Inc.

Danilo Ardagna and Barbara Pernici. 2005. Global and Local QoS Guarantee in Web Service Selection.. In *BPM Workshops*.

Luciano Baresi and Sam Guinea. 2011. Self-Supervising BPEL Processes. *IEEE Transactions on Software Engineering* 37, 2 (2011), 247–263.

Johan Bengtsson and Wang Yi. 2003. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets (Lecture Notes in Computer Science)*, Vol. 3098. Springer, 87–124.

Domenico Bianculli, Dimitra Giannakopoulou, and Corina S. Pasareanu. 2011. Interface decomposition for service compositions. In *ICSE*. 501–510.

Milan Ceska, Frits Dannenberg, Marta Z. Kwiatkowska, and Nicola Paoletti. 2014. Precise Parameter Synthesis for Stochastic Biochemical Systems. In *CMSB (Lecture Notes in Computer Science)*, Pedro Mendes, Joseph O. Dada, and Kieran Smallbone (Eds.), Vol. 8859. Springer, 86–98. DOI:http://dx.doi.org/10.1007/978-3-319-12982-2_7

Manman Chen, Tian Huat Tan, Jun Sun, Yang Liu, and Jin Song Dong. 2014. VeriWS: a tool for verification of combined functional and non-functional requirements of web service composition. In *ICSE*. 564–567. DOI:http://dx.doi.org/10.1145/2591062.2591070

Manman Chen, Tian Huat Tan, Jun Sun, Yang Liu, Jun Pang, and Xiaohong Li. 2013. Verification of Functional and Non-functional Requirements of Web Service Composition. In *ICFEM*. 313–328. DOI: http://dx.doi.org/10.1007/978-3-642-41202-8_21

Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. 2007. Web Services Description Language (WSDL) Version 2.0. W3C Recommendation, available at http://www.w3.org/TR/wsdl20/. (June 2007).

Alessandro Cimatti, Luigi Palopoli, and Yusi Ramadian. 2008. Symbolic Computation of Schedulability Regions Using Parametric Timed Automata. In *RTSS*. IEEE Computer Society, 80–89. DOI: http://dx.doi.org/10.1109/RTSS.2008.36

Conrado Daws and Sergio Yovine. 1996. Reducing the number of clock variables of timed automata. In *RTSS*. IEEE Computer Society, 73–81.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science)*. Springer, 337–340.

Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. 2007. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In *WCET*.

Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. 2011. Run-time efficient probabilistic model checking. In *ICSE*. 341–350. DOI: http://dx.doi.org/10.1145/1985793.1985840

Howard Foster. 2006. *A Rigorous Approach To Engineering Web Service Compositions*. Ph.D. Dissertation. Imperial College of London.

Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. 2006. LTSA-WS: a tool for model-based verification of Web service compositions and choreography. In *ICSE*. 771–774.

Apache Software Foundation. 2007. Apache ODE. http://ode.apache.org/. (2007).

Laurent Fribourg, David Lesens, Pierre Moro, and Romain Soulat. 2012. Robustness Analysis for Scheduling Problems using the Inverse Method. In *TIME*. IEEE Computer Society Press, 73–80. DOI: http://dx.doi.org/10.1109/TIME.2012.10

Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. 2007. Simple Object Access Protocol (SOAP) Version 1.2. W3C Recommendation, available at http://www.w3.org/TR/soap12/. (april 2007).

Aleksandra Jovanović and Marta Z. Kwiatkowska. 2014. Parameter Synthesis for Probabilistic Timed Automata Using Stochastic Game Abstractions. In *RP (Lecture Notes in Computer Science)*, Joël Ouaknine, Igor Potapov, and James Worrell (Eds.), Vol. 8762. Springer, 176–189. DOI: http://dx.doi.org/10.1007/978-3-319-11439-2_14

Stephan Kraft, Sergio Pacheco-Sanchez, Giuliano Casale, and Stephen Dawson. 2009. Estimating service resource consumption from response time measurements. In *VALUETOOLS*. 48.

Alfons Laarman, Mads Chr. Olesen, Andreas Engelbredt Dalsgaard, Kim Guldstrand Larsen, and Jaco Van De Pol. 2013. Multi-Core Emptiness Checking of Timed Büchi Automata using Inclusion Abstraction. In *CAV (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 968–983.

Thi Thieu Hoa Le, Luigi Palopoli, Roberto Passerone, Yusi Ramadian, and Alessandro Cimatti. 2010. Parametric analysis of distributed firm real-time systems: A case study. In *ETFA*. IEEE, 1–8.

Yi Li, Tian Huat Tan, and Marsha Chechik. 2014. Management of Time Requirements in Component-Based Systems. In *FM*. 399–415.

Jeff Magee and Jeff Kramer. 2006. *Concurrency - state models and Java programs (2. ed.)*. Wiley. I–XVIII, 1–413 pages.

Daniel A. Menascé. 2004. Response-Time Analysis of Composite Web Services. *IEEE Internet Computing* 8, 1 (2004), 90–92.

Oliver Moser, Florian Rosenberg, and Schahram Dustdar. 2008. Non-intrusive monitoring and service adaptation for WS-BPEL. In *WWW*. 815–824.

Gregory J. Pottie and William J. Kaiser. 2000. Wireless Integrated Network Sensors. *Commun. ACM* 43, 5 (2000), 51–58.

Eric Schmieders and Andreas Metzger. 2011. Preventing Performance Violations of Service Compositions Using Assumption-Based Run-Time Verification. In *Towards a Service-Based Internet - 4th European Conference, ServiceWave 2011, Poznan, Poland, October 26-28, 2011. Proceedings*. 194–205. DOI: http://dx.doi.org/10.1007/978-3-642-24755-2_19

Alexander Schrijver. 1986. *Theory of linear and integer programming*. John Wiley and Sons.

Jocelyn Simmonds, Shoham Ben-David, and Marsha Chechik. 2010. Guided recovery for Web service applications. In *SIGSOFT FSE*. 247–256.

Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. 2013. Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP. *ACM Transactions on Software Engineering and Methodology* 22, 1 (2013), 3.1–3.29. DOI: http://dx.doi.org/10.1145/2430536.2430537

Tian Huat Tan, Étienne André, Manman Chen, Jun Sun, Yang Liu, and Jin Song Dong. 2015. SELAMAT: Tool For Synthesis Local Time Requirement. https://sites.google.com/site/automatedsynthesis/home/. (2015).

Tian Huat Tan, Étienne André, Jun Sun, Yang Liu, Jin Song Dong, and Manman Chen. 2013. Dynamic synthesis of local time requirement for service composition. In *ICSE*. 542–551.

Tian Huat Tan, Manman Chen, Étienne André, Jun Sun, Yang Liu, and Jin Song Dong. 2014. Automated runtime recovery for QoS-based service composition. In *WWW*. 563–574. DOI: http://dx.doi.org/10.1145/2566486.2568048

Louis-Marie Traonouez, Didier Lime, and Olivier H. Roux. 2009. Parametric Model-Checking of Stopwatch Petri Nets. *Journal of Universal Computer Science* 15, 17 (2009), 3273–3304.

Tao Yu, Yue Zhang, and Kwei-Jay Lin. 2007. Efficient algorithms for Web services selection with end-to-end QoS constraints. *TWEB* 1, 1 (2007).