Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems | School of Computing and Information Systems

7-2020

Automated synthesis of local time requirement for service composition

Étienne ANDRÉ

Tian Huat TAN

Manman CHEN

Shuang LIU

Jun SUN
Singapore Management University, junsun@smu.edu.sg

See next page for additional authors

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Citation

ANDRÉ, Étienne; TAN, Tian Huat; CHEN, Manman; LIU, Shuang; SUN, Jun; LIU, Yang; and DONG, Jin Song. Automated synthesis of local time requirement for service composition. (2020). *Software and Systems Modeling*. 19, 983-1013.
Available at: https://ink.library.smu.edu.sg/sis_research/5882

This Journal Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Author
Étienne ANDRÉ, Tian Huat TAN, Manman CHEN, Shuang LIU, Jun SUN, Yang LIU, and Jin Song DONG

This journal article is available at Institutional Knowledge at Singapore Management University:
https://ink.library.smu.edu.sg/sis_research/5882

Software and Systems Modeling (2020) 19:983–1013
https://doi.org/10.1007/s10270-020-00787-5

REGULAR PAPER

Automated synthesis of local time requirement for service composition

Étienne André[1] · Tian Huat Tan[2] · Manman Chen[3] · Shuang Liu[4] · Jun Sun[5] · Yang Liu[6] · Jin Song Dong[7,8]

Received: 20 February 2018 / Revised: 26 February 2020 / Accepted: 27 February 2020 / Published online: 13 March 2020
© Springer-Verlag GmbH Germany, part of Springer Nature 2020

Abstract

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 use the BPEL modeling language that aims at specifying Web services. We extend it with timing parameters and equip it with a formal semantics. Then, we propose a fully automated approach to synthesize the response time requirement of component services modeled using BPEL, 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.

Keywords Web service composition · Parameter synthesis · Modeling Web services · Formal semantics · BPEL · Parametric model checking

Communicated by Gary Leavens.

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"). Étienne André and Jun Sun are partially supported by the ANR-NRF French-Singaporean research program ProMiS (ANR-19-CE25-0015).

✉ Étienne André
eandre93430@lipn13.fr

1 CNRS, Inria, LORIA, Université de Lorraine, Nancy, France

2 IBM, Singapore, Singapore

3 Autodesk, Singapore, Singapore

4 College of Intelligence and Computing, Tianjin University, Tianjin, China

5 Singapore Management University, Singapore, Singapore

6 Nanyang Technological University, Singapore, Singapore

7 National University of Singapore, Singapore, Singapore

8 Griffith University, Brisbane, Australia

1 Introduction and motivation

Service-oriented architecture is a paradigm where building blocks are used as services for software applications. Services encapsulate their functionalities and information and make them available through a set of operations accessible over a network infrastructure using standards like SOAP [39] and WSDL [25]. To make use of a set of services to achieve a business goal, service composition languages such as BPEL (Business Process Execution Language) [6] 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 refer to the response time requirement of composite services as *global time requirement*, and to 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 component services when building a composite service while satisfying the response time requirement.

An additional motivation for our work is that of microservices. As pointed out by [74], many big players in the market (e.g., Netflix, Amazon and Microsoft Azure) have adopted microservice architecture [57] by decomposing their existing monolithic applications into smaller, and highly decoupled services (also known as microservices). These services are then composed for fulfilling their business requirements. For example, Netflix decomposed their monolithic DVD rental application into services that work together and that stream digital entertainment to millions of Netflix customers every day. Services of Netflix are hosted in a cloud provided by Amazon EC2 [7], which offers about 40 instance types. The problem of composition of Web services with a large set of microservices is more and more relevant now, as the microservices are getting more popular than ever (see e.g., [56,67]). This justifies the use of techniques for which different services can be compared to and eventually selected. Service-oriented architecture and microservice architectures are conceptually similar: service-oriented architecture is a term that is used earlier and also widely used in literature. Microservice architecture is more of a newer term that is used and practice widely in current industry, for the purpose of agile development. (For detailed comparison, see e.g., [21].) The methods developed here are applicable to both service-oriented architecture and microservice architecture.

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 3 s 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 maintain as many combinations of 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 perform a rigorous model-based analysis of Web services, in order to synthesize the local time requirement in composite services. Our approach performs an analysis of the composite service model behavior, using techniques inspired by parameter synthesis for timed systems. Our synthesis approach does not only avoid bad scenarios in the service composition, but also guarantees the fulfillment of the global time requirement.

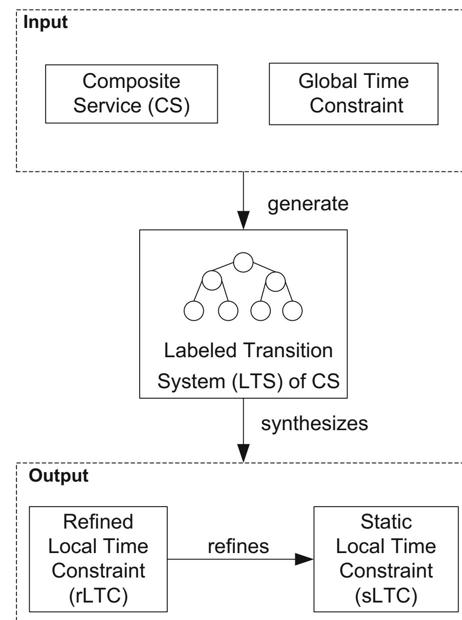


Fig. 1 General approach

We use as a formalism BPEL, which is a *de facto* standard language for specifying 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. More precisely, response times of component services can be dependent; therefore, constraint between services may be of the form, e.g., $t_{fs} > t_{hs}$ (for two parametric component service times), rendering the problem quite delicate. In addition, there may be multiple possibilities of component services' response times that are satisfiable. This can be particularly delicate to find out with only manual human inspection.

Figure 1 illustrates the main steps of our approach for synthesizing local time requirements. The required inputs are the specification of the composite service 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.

We first propose a formal 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 in the form of a labeled transition system (LTS).

Based on the LTS resulting from the input composite service, we then 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 of the model, since it is unknown which execution path will be executed at runtime. (This will depend on the dynamic behavior of the system.) 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, when creating a new composite service, it allows the selection of feasible services 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 might have been 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.

Due to the highly evolving and dynamic environment which 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. We refer to the sLTC refined at runtime as the *refined local time constraint* (hereafter rLTC). The rLTC is used to decide whether the current composite service can still satisfy the global time requirement, despite some unplanned issues such as network congestion.

Our contributions are summarized as follows.

1. We augment the BPEL modeling language with timing parameters, and we equip it with a formal semantics in the form of a labeled transition system.
2. Given a composite service modeled using BPEL, we develop a sound method for synthesizing the local time requirement in the form of a set of constraints, which can be applied at the design stage of service composition.
3. We introduce a refinement procedure on the sLTC of a composite service based on the runtime information, which results in a more relaxed rLTC. The rLTC can be used to verify whether the composite service could still eventually satisfy the global time requirement at runtime.
4. We implement our algorithms into a tool SELAMAT. We then conduct experiments on several examples. The results show that the rLTC can indeed help to improve the accuracy of the sLTC. In addition, we show that the runtime adaptation does not incur much overhead in practice.

1.2 About this manuscript

This manuscript is an extended version of [72]. We in fact rewrote most of the manuscript for a better readability. The most notable differences between this manuscript and [72] are:

1. we replaced the formerly defined "AOLTS" with what we believe to be a simpler and more elegant presentation of labeled transition systems (LTS);
2. we added details on our implementation and used more service composition examples; and,
3. most importantly, we added a refinement procedure that attempts to meet the global time requirement at runtime even when the constraint computed statically is violated (Sect. 6).

1.3 Outline

The rest of this paper is structured as follows. Section 2 introduces a timed BPEL running example. Section 3 provides the necessary definitions and terminologies. Section 4 introduces our formal semantics for BPEL extended with timing parameters. Section 5 presents the synthesis algorithms for sLTC. Section 6 introduces rLTC and its usage for runtime adaptation of a service composition. Section 7 evaluates our approach with four service composition examples. Section 8 reviews related works. Finally, Sect. 9 concludes the paper and outlines future work.

2 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 3 s upon request.

SMIS has three component Web services, i.e., 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

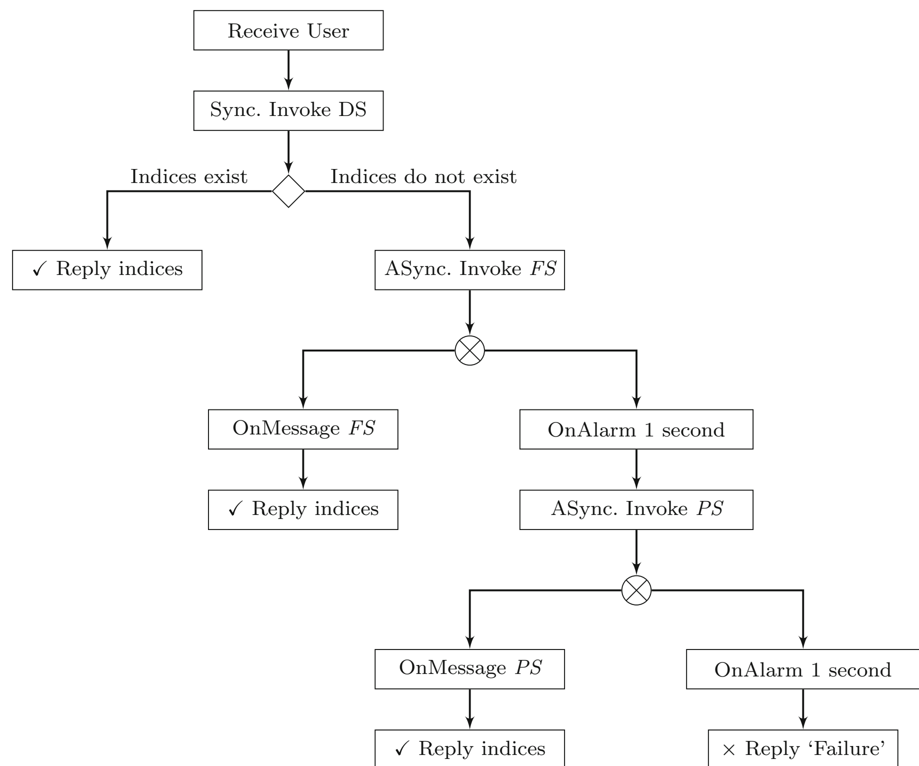


Fig. 2 Stock Market Indices Service

here). The workflow of the SMIS is sketched in Fig. 2 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 reaches an `<if>` branch (denoted by $\Diamond$). 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>` construct (denoted by $\otimes$) is used here to await an incoming response (`<onMessage>`) from previous asynchronous invocation or timeout (`<onAlarm>`), whichever occurs. If the response from FS (`OnMessage FS`) is received within 1 s, then the result is returned to the user (`Reply indices`). Otherwise, the timeout occurs (`OnAlarm 1 second`), then 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 1 s. Otherwise, it notifies the user regarding the failure of getting stock indices (`Reply 'Failure'`). The states marked with a $\checkmark$ (resp. $\times$) represent desired (resp. undesired) end states.

The global time requirement for SMIS is that SMIS should respond within 3 s 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 can also help to choose a paid service PS which is both cheap and responds quickly enough.

In this example, an activity to avoid (which will be referred to as a "bad activity" in the following) is the reply activity that is triggered after the component service PS fails to respond within 1 s, which is marked with $\times$ in Fig. 2.

3 A formal model for parametric composite services

3.1 Variables, clocks, parameters and constraints

Given a finite set $\mathcal{V}$ of finite-domain *variables*, 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 $v(y) = \bot$ 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 (parametric) timed automata [4,5] and (parametric) stateful timed CSP [11,68]. Let $X = \{x_1, \dots, x_h\}$ (for some integer h) be a finite set of clocks, i.e., real-valued variables evolving at the same rate. A *clock valuation* is a function $w : X \to \mathbb{R}_{\geq 0}$ that assigns a nonnegative real value to each clock.

Let $\Lambda = \{\lambda_1, \dots, \lambda_m\}$ (for some integer m) be a finite set of *parameters*, i.e., rational-valued constants that will be used

here to represent the unknown response time of a component service. A *parameter valuation* is a function $\pi : \Lambda \rightarrow \mathbb{Q}_{\geq 0}$ assigning a nonnegative rational value to each parameter.

Henceforth, we use w (resp. π) to denote a clock (resp. parameter) valuation.

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

A *convex constraint* (or *constraint*) is a conjunction of inequalities. We denote by $\mathcal{C}_{X \cup \Lambda}$ the set of all convex constraints over X and Λ. Similarly, we denote by $\mathcal{C}_\Lambda$ the set of all convex constraints over Λ.

Let $C \in \mathcal{C}_{X \cup \Lambda}$, $C[\pi]$ denotes the valuation of C with π, i.e., the constraint over X obtained by replacing each $\lambda \in \Lambda$ with $\pi(\lambda)$ in C. Note that $C[\pi]$ can be written as $C \wedge \bigwedge_{\lambda_i \in \Lambda} \lambda_i = \pi(\lambda_i)$. We say that w satisfies $C[\pi]$ if the expression obtained by replacing each $x \in X$ in $C[\pi]$ with $w(x)$ evaluates to true.

Given $C \in \mathcal{C}_{X \cup \Lambda}$, we define $C^{\uparrow}$ as the *time elapsing* of C, i.e., the constraint over X and Λ obtained from C by delaying all clocks by an arbitrary amount of time. That is:

$$C^{\uparrow} = \{(w', \pi) \mid w \text{ satisfies } C[\pi] \wedge \forall x \in X : \\ w'(x) = w(x) + d, d \in \mathbb{R}_{\geq 0}\}.$$

Given $C \in \mathcal{C}_{X \cup \Lambda}$ and $X' \subseteq X$, we denote by $prune_{X'}(C)$ the constraint in $\mathcal{C}_{X \cup \Lambda}$ 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., [65]). More generally, given $C \in \mathcal{C}_{X \cup \Lambda}$, we denote by $C{\downarrow_\Lambda}$ the *projection* of constraint C onto Λ, i.e., the constraint obtained from C by pruning all clock variables. Again, such a projection can be computed using Fourier–Motzkin elimination.

A *non-necessarily convex constraint* (or NNCC) is a conjunction of disjunction of inequalities[1]; NNCCs are used 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 $\mathcal{NC}_\Lambda$ the set of all NNCCs over Λ.

Given $C \in \mathcal{NC}_\Lambda$, 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*. Given two constraints $C_1, C_2 \in \mathcal{NC}_\Lambda$, we say that C_2 is *weaker* (or *more relaxed*) than C_1, denoted by $C_1 \subseteq C_2$, if $\forall \pi : \pi \models C_1 \Rightarrow \pi \models C_2$.

[1] Without loss of generality, we assume here that all NNCCs are in conjunctive normal form (CNF).

3.2 Syntax of composite service processes

BPEL [6] 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. These activities are *communication activities*. The control flow of the service is defined using structural activities such as `<flow>`, `<sequence>`, `<pick>` and `<if>`.

A composite service CS makes use of a finite number of component services to accomplish a task. Let $E = \{\mathsf{S}_1, \ldots, \mathsf{S}_n\}$ 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 the following.

Definition 1 *Processes* are defined using the following grammar:

$P \mathrel{\widehat{=}}$	$rec(\mathsf{S})$	receive activity
$\mid$	$reply(\mathsf{S})$	reply activity
$\mid$	$sInv(\mathsf{S})$	synchronous invocation
$\mid$	$aInv(\mathsf{S})$	asynchronous invocation
$\mid$	$P \mid\mid\mid Q$	concurrent activity
$\mid$	$P \,;\, Q$	sequential activity
$\mid$	$P \lhd b \rhd Q$	conditional activity
$\mid$	$pick(\biguplus_{i=1}^{n} \mathsf{S}_i \Rightarrow P_i, \biguplus_{j=1}^{k} alrm(a_j) \Rightarrow Q_j)$	pick activity

where S is a component service, P and Q are composite service processes, b is a Boolean expression and $a_j \in \mathbb{Q}_{>0}$ are positive rational numbers, for $1 \leq j \leq k$.

Let us describe below the BPEL syntax notations introduced in Definition 1:

- $rec(\mathsf{S})$ and $reply(\mathsf{S})$ are used to denote “receive from” and “reply to” a service S, respectively;
- $sInv(\mathsf{S})$ (resp. $aInv(\mathsf{S})$) 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;
- $P \,;\, Q$ denotes the sequential composition of BPEL activities P and Q;

[2] We discuss the time incurred for internal operations in Sect. 6.6.

- $P \lhd b \rhd Q$ 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;
- $pick(\biguplus_{i=1}^{n} \mathsf{S}_i \Rightarrow P_i, \biguplus_{j=1}^{k} alrm(a_j) \Rightarrow Q_j)$ denotes the BPEL *pick* composition, which contains two types of activities, i.e., *onMessage* activity and *onAlarm* activity. An *onMessage* activity $\mathsf{S}_i \Rightarrow P_i$ is activated when the message from service S_i arrives and BPEL activity P_i is subsequently executed; an *onAlarm* activity $alrm(a_j) \Rightarrow Q_j$ is activated at a_j time units, and BPEL activity Q_j 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 are activated at the same time, one of the activities will be chosen non-deterministically for execution. Given a *pick* activity P, we use $P.onMessage$ and $P.onAlarm$ 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.

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 $sInv(\mathsf{S})$, it requires an input variable v_i that specifies the date information and an output variable v_o to hold the return value from $sInv(\mathsf{S})$. To keep the notations concise, we abstract the usage and assignment of variables for communication activities.

We make the following assumption throughout this manuscript:

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 Sect. 6.6 for a discussion).

3.3 Parametric composite service models

Let us now formally define composite service models and parametric composite service models. Let $\mathcal{P}_{np}$ denote the set of all possible (nonparametric) composite service processes.

Definition 2 (*Composite service model*) A composite service model CS is a tuple $(\mathcal{V}, v_0, N_0)$, where $\mathcal{V}$ is a finite set of variables, $v_0 \in Valuations(\mathcal{V})$ 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 Definition 1) making use of the variables in $\mathcal{V}$.

Each service comes with a *response time*, which is a rational-valued constant, and can be seen as an upper bound on the time that a service needs to successfully return its answer.

Given a composite service CS, let $t_i \in \mathbb{R}_{\geq 0}$ be the response time of component service S_i for $i \in \{1, \dots, n\}$, and let $E_t = \{t_1, \dots, t_n\}$ be a set of component service response times that fulfill the global time requirement of service CS. Because t_i, for $i \in \{1, \dots, n\}$, is a rational 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*.

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 S_i, the response time of which is now a parameter $\lambda_i \in \Lambda$, instead of a rational-valued constant. Then, a parametric composite service process is a service process defined according to the grammar of Definition 1, where services ("S" in Definition 1) 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 3 (*Parametric composite service model*) A parametric composite service model CS is a tuple $(\mathcal{V}, v_0, \Lambda, P_0, C_0)$, where $\mathcal{V}$ is a finite set of variables; $v_0 \in Valuations(\mathcal{V})$ is an initial valuation that maps each variable to its initial value; Λ 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 $C_0 \in \mathcal{C}_\Lambda$ is the initial parametric constraint.

Example 1 Let $\mathcal{V} = \{y_1\}$. Let v_0 be such that $v_0(y_1) = 0$. Let $\Lambda = \{\lambda_1, \lambda_2\}$. Let $P_0 = pick(\mathsf{S} \Rightarrow sInv(\mathsf{S}_1), alrm(1) \Rightarrow sInv(\mathsf{S}_2)) \lhd y_1 > 0 \rhd Stop$. Let $C_0 = \lambda_1 < \lambda_2$. Let λ_i denote the response time of $sInv(\mathsf{S}_i)$, $i \in \{1, 2\}$.

Then $\mathsf{CS} = (\mathcal{V}, v_0, \Lambda, P_0, C_0)$ is a parametric composite service model.

Process and model valuation Given a parametric composite service process P with a parameter set $\Lambda = \{\lambda_1, \dots, \lambda_m\}$ and given a parameter valuation $(\pi(\lambda_1), \dots, \pi(\lambda_m))$, $P[\pi]$ denotes the *valuation* of P with π, i.e., the process where each occurrence of a parameter λ_i is replaced with its valuation $\pi(\lambda_i)$.

Given a parametric composite service model CS with a parameter set $\Lambda = \{\lambda_1, \dots, \lambda_m\}$, and given a parameter

valuation $(\pi(\lambda_1), \ldots, \pi(\lambda_m))$, $\mathsf{CS}[\pi]$ denotes the *valuation* of CS with π, i.e., the model $(\mathcal{V}, v_0, \Lambda, P_0, C)$, where C is $C_0 \wedge \bigwedge_{i=1}^{m}(\lambda_i = \pi(\lambda_i))$. Note that $\mathsf{CS}[\pi]$ can be seen as a nonparametric service model $(\mathcal{V}, v_0, P_0[\pi])$.

Example 2 Consider the parametric composite service model CS defined in Example 1. Assume π such that $\pi(\lambda_1) = 1$ and $\pi(\lambda_2) = 2$. Then $P_0[\pi] = pick(\mathsf{S} \Rightarrow sInv(\mathsf{S}_1), alrm(1) \Rightarrow sInv(\mathsf{S}_2)) \lhd y_1 > 0 \rhd Stop$, where the response time of $sInv(\mathsf{S}_1)$ is 1, and the response time of $sInv(\mathsf{S}_2)$ is 2.

3.4 Bad activities

Given a BPEL service CS, we define a *bad activity* as an atomic activity such that its execution leads the composite service CS to violate 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 attributes of BPEL activities. This work can be performed manually or using semi-automated procedures.

Example 3 Consider again the example in Sect. 2. Then "Reply 'Failure'" is a bad activity, denoted in Fig. 2 by $\times$.

4 A formal semantics for parametric composite services

In this section, we provide our parametric composite service model with a formal semantics, defined 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) [11,68], that makes use of implicit clocks.

We first recall LTSs (Sect. 4.1) and define symbolic states (Sect. 4.2). Following that, we define implicit clocks and the associated functions, i.e., activation and idling (Sect. 4.3). We then introduce our formal semantics (Sect. 4.4) and apply it to an example (Sect. 4.5). We finally prove a technical result relating the nonparametric and the parametric service models (Sect. 4.6).

4.1 Labeled transition systems

Definition 4 (*Labeled transition system*) A labeled transition system (LTS) is a tuple $LTS = (S, s_0, \Sigma, \delta)$, where

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

Given $LTS = (S, s_0, \Sigma, \delta)$, a state $s \in S$ is a *terminal state* if there does not exist a state $s' \in S$ 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 $s, s' \in S$, 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 $s_i \in S$ 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 $s \in S$, we use $succ(s)$ to denote the set of states reachable in one step from s; formally, $succ(s) = \{s' \mid \exists a \in \Sigma, \exists s' \in S : (s, a, s') \in \delta\}$.

In the following, we introduce the notion of LTS starting from a state s which is defined as the LTS containing s and all its successor states and transitions.

Definition 5 (*sub-LTS*) Let $LTS = (S, s_0, \Sigma, \delta)$ be an LTS, and let s be a state of S. The sub-LTS of LTS starting from s is $(S', s, \Sigma', \delta')$, where

1. $S' \subseteq S$ is the set of states reachable from $s \in S$ in LTS;
2. $\delta' \subseteq \delta$ is the transition relation satisfying the following condition: $(s_1, a, s_2) \in \delta'$ if $s_1, s_2 \in S'$ and $(s_1, a, s_2) \in \delta$; and
3. $\Sigma' \subseteq \Sigma$ is the set of all actions used in δ', i.e., $\{a \mid \exists s_1, s_2 \in S' : (s_1, a, s_2) \in \delta'\}$.

4.2 Symbolic states

In the following, we equip our parametric composite service models with a symbolic semantics, i.e., a semantics, a run of which will capture a (possibly infinite) set of runs, for a (possibly infinite) set of parameter valuations.

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

Definition 6 (*State*) Given a parametric composite service model $\mathsf{CS} = (\mathcal{V}, v_0, \Lambda, P_0, C_0)$, 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 $\mathcal{C}_{X \cup \Lambda}$, and $D \in \mathcal{L}_{\Lambda}$ 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 field 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 $s \in S$ by (P, C, D) for the sake of brevity.

4.3 Implicit clocks

In order to provide parametric composite service models with a symbolic semantics, we use *clocks* to record the elapsing of

time. Recall from Sect. 3.1 that clocks are real-valued variables initially equal to 0, and evolving all at the same rate; some clocks may be reset to 0. Clocks are used to record the time elapsing in several formalisms, in particular in timed automata (TAs) [4]. 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 [17]. In (P)TAs, it is possible to dynamically reduce the number of clocks [8,28]. An alternative approach is to define a semantics that create clocks on the fly when necessary, and prune them when they are no longer needed. This approach was initially proposed for stateful timed CSP [68] and shares similarities with *firing times* in time Petri nets [54]. This allows a smaller state space compared to the explicit clock approach. We refer to this second approach [68] as the *implicit clock approach* and adopt this implicit clock approach in our work.

4.3.1 Clock activation

Clocks are implicitly associated with processes. For instance, given a communication activity $sInv(\mathsf{S})$, a clock starts measuring time once the activity becomes activated. To introduce clocks on the fly, we define an activation function *Act* in the following definition, in the spirit of the one defined in [11,68].

In short, this definition explains how to associate a new clock with a process: this clock will only be associated with the new processes with timing constraints, while it will not be associated with untimed processes nor to processes to which another implicit clock is already associated.

Definition 7 Given a process, we define the activation function *Act* using the following set of recursive rules:

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

where $A \in \{rec, sInv, aInv, reply\}$, $\oplus \in \{|||, \lhd b \rhd\}$, and $mpick = pick(\biguplus_{i=1}^{n} \mathsf{S}_i \Rightarrow P_i, \biguplus_{j=1}^{k} alrm(a_j) \Rightarrow Q_j)$

Let us explain Definition 7. 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 activate the child activities for BPEL structural activities.
- For rule A6, function *Act* is applied only to activity P, but not to activity Q, since activity P is the immediate subsequent activity. (Activity Q will be executed only after the completion of activity P.)

Example 4 Let $P = sInv(\mathsf{S}_1) \;|||\; aInv(\mathsf{S}_2)$. Then, applying rules A5 and A1, $Act(P, x) = sInv(\mathsf{S}_1)_x \;|||\; aInv(\mathsf{S}_2)_x$. Note that x is associated with both processes, as they are both simultaneously activated.

Example 5 Let $P = sInv(\mathsf{S}_1)_{x'}\,; aInv(\mathsf{S}_2)$. Then, applying rules A6 and A3, $Act(P, x) = sInv(\mathsf{S}_1)_{x'}\,; aInv(\mathsf{S}_2)$. Indeed, the first invocation $sInv(\mathsf{S}_1)_{x'}$ is already associated with another clock x' (rule A3) while the right-hand part of the sequence is not yet activated (rule A6).

Given a process P, we denote by $aclk(P)$ the set of *active clocks* associated with P.

Example 6 Assume process $P = sInv(\mathsf{S}_1)_{x_0} \;|||\; sInv(\mathsf{S}_2)_{x_1}$. The set of active clocks associated with P is $aclk(P) = \{x_0, x_1\}$.

4.3.2 Idling function

We define in Definition 8 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 $X \cup \Lambda$. This idling function is similar in essence to the *time elapsing* on symbolic states (zones or parametric zones) defined for TAs or PTAs [17,41].

Definition 8 Given a process, we define the idling function *idle* using the following set of recursive rules:

$$\begin{array}{lcll} idle(A(\mathsf{S})_x) & = & x \le \lambda_{\mathsf{S}} & \text{I1}\\ idle(B(\mathsf{S})_x) & = & x = 0 & \text{I2}\\ idle(P\oplus Q) & = & idle(P)\wedge idle(Q) & \text{I3}\\ idle(P\,;Q) & = & idle(P) & \text{I4}\\ idle(mpick_x) & = & x \le \lambda_{\mathsf{S}} \wedge \bigwedge_{j=0}^{k} x \le a_j & \text{I5} \end{array}$$

where $A \in \{rec, sInv\}$, $B \in \{aInv, reply\}$, $\oplus \in \{|||, \lhd b \rhd\}$, $mpick = pick(\biguplus_{i=1}^{n} \mathsf{S}_i \Rightarrow P_i, \biguplus_{j=1}^{k} alrm(a_j) \Rightarrow Q_j)$, and λ_{S} is the parametric response time of service $mpick_x$.

Let us explain Definition 8.

- 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 λ_S of the service: that is, one can only remain in this state while $x \leq \lambda_S$ remains valid.
- Rule I2 considers the situation when no waiting is required: therefore, the clock constraint $x = 0$ implies that this state should be left within 0-time, as these actions are instantaneous.
- Rules I3 and I4 state that the function *idle* is applied recursively to activate the 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.
- Rule I5 states that the activity can idle only until λ_S or any of the alarms a_j is reached. The conjunction comes from the fact that, as soon as any alarm reaches its timeout, then it will be triggered, therefore leading the system to leave this symbolic state.

Example 7 Let $P = sInv(\mathsf{S}_1) \;|||\; aInv(\mathsf{S}_2)$. Assume the response time of S_i is λ_i for $i \in \{1, 2\}$. Recall from Example 4 that $Act(P, x) = sInv(\mathsf{S}_1)_x \;|||\; aInv(\mathsf{S}_2)_x$. Let us apply *idle* to $Act(P, x)$. Applying rules I3, I1 and I2, we get $x \leq \lambda_1 \wedge x = 0$.

4.4 Operational semantics

The operational semantics will be defined in the form of an LTS. The actions labeling the LTS will be sequences of rules; these rules will be a set of rules (similar to those of parametric stateful timed CSP [11]) defining the transitions of the semantics and will be explained below. Let

$$\begin{aligned}\mathsf{Rules} = \{&rSInv, rRec, rReply, rAInv, rCond1, rCond2,\\ &rCond3, rCond4, rSeq1, rSeq2, rFlow1,\\ &rFlow2, \} \cup (rPickM \times \mathbb{N}) \cup (rPickA \times \mathbb{N})\end{aligned}$$

be the set of rules that will be used by the LTS. Two rules (*rPickM* and *rPickA*) are associated with a positive integer, so as to remember which subprocess is derived (this will be explained later on). Let Sequences(Rules) denote the set of *sequences* of rules, i.e., non-empty ordered elements of Rules (possibly used several times). An example of a sequence of rule is $\langle rRec, rReply, rRec, (rPickM, 2)\rangle$. Sequence concatenation is denoted by operator $+$.

We can now define the semantics of a parametric composite service model in the form of an LTS. Let $ClkSeq = \langle x_0, x_1, \ldots\rangle$ be a sequence of clocks. We will need *ClkSeq* to pick a fresh clock when applying the clock activation function *Act* defined previously.

Definition 9 (*Semantics of composite services*) Let $\mathsf{CS} = (\mathcal{V}, v_0, \Lambda, P_0, C_0)$ be a parametric composite service model. The semantics of CS (hereafter denoted by LTS_{CS}) is the LTS $(S, s_0, \mathsf{Sequences(Rules)}, \delta)$ where

$$S = \{(v, P, C, D) \in Valuations(\mathcal{V}) \times \mathcal{P} \times \mathcal{C}_{X\cup\Lambda} \times \mathcal{L}_\Lambda\},$$
$$s_0 = (v_0, P_0, C_0, 0)$$

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

The transition relation $\hookrightarrow$ is specified by a set of rules, given in "Appendix A." Let us first explain these rules, after which we will go back to the explanation of Definition 9. The transition relation is labeled by a sequence of rules that allows one to remember by using which sequence of rules a process evolves into another one.

Synchronous invocation Rule *rSInv* states that a state $s = (v, sInv(\mathsf{S})_x, C, D)$ may evolve into the state $s' = (v', Stop, (x = \lambda_S) \wedge C^{\uparrow}, D + \lambda_S)$, where *Stop* is the activity that does nothing, and λ_S is the parametric response time of component service S. Note that, from Definition 9, the condition $(x = \lambda_S) \wedge C^{\uparrow}$ is necessarily satisfied (otherwise this evolution is not possible). Furthermore, the parametric duration from the initial state (D) is incremented by λ_S. Rules *rRec*, *rReply* and *rAInv* are similar.

Pick activity Rule *rPickM* encodes the transition that takes place due to an *onMessage* activity, where λ_i denotes the parametric response time of P_i. Let us explain the constraint $(x = \lambda_i) \wedge idle(mpick_x) \wedge C^{\uparrow}$. First, after the transition, the current clock x needs to be equal to the parametric response time of service S_i, i.e., $x = \lambda_i$. 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. Observe that the transition in $\hookrightarrow$ is labeled using the pair $(rPickM, i)$ so as to remember that the ith process (i.e., P_i) has been selected.

Rule *rPickA* (for an *onAlarm* activity) is similar; observe that, instead of using the parametric response time, we use the time stipulated by the alarm (i.e., a_j) of process Q_j.

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 $v(b) \in \{true, false, \perp\}$. We have that $v(b) = \perp$ 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 at runtime, we explore both activities A and B when $v(b) = \perp$ so as to reason about all possible scenarios. The case of $v(b) = \perp$ 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 $A\,;B$ may evolve into a state containing activity $A'\,;B$. Otherwise, if A is a *Stop* activity (i.e., activity A has finished its execution), then the state may evolve into B. This is captured by *rSeq*2.

Concurrent activity For concurrent activity $A \;|||\; B$, both activities A and activity B are 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'), then a state containing $A \;|||\; B$ can evolve into a state containing $A' \;|||\; B$, if $C' \wedge idle(B)$ holds. That is, the clock constraints in C' cannot exceed the duration activity B can last for. Rule *rFlow*2 is dual.

Let us now explain Definition 9. 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 *ClkSeq*. 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'. More in detail, the expression $prune_{X \setminus aclk(P')}(C')$ denotes that we remove all clocks from the obtained constraint C' by existential quantification, except those which are still active in the successor P' of P (recall that $prune_X(C)$ was defined in Sect. 3.1).

Observe that one clock is introduced, and zero or more clocks may be pruned during a transition. In practice, a clock is introduced only when necessary; if the activation function does not activate any subprocess, no new clocks are created.

Good and bad states Let us define good and bad states in the LTS obtained from Definition 9. The execution of a bad activity will make the execution of CS end in an undesired terminal state, which we refer to as a *bad state*. A terminal state which is not a bad state is called a *good state*.

4.5 Application to an example

Consider a composite service CS starting from $pick(\mathrm{PS} \Rightarrow reply(User), alrm(1) \Rightarrow [reply(User)]_{bad})$. Assume λ_{PS} is the parametric response time of service PS. (Note that CS is a part of the SMIS example from Sect. 2.) The states of CS computed according to Definition 9 are given in Fig. 3, including intermediate states (detailed in the following). Since CS has no variable, then $v = \varnothing$ in all states; therefore, we omit the component v from all states for 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 the intermediate state s_0^x, the process may evolve into the intermediate state s_1' by applying the rule *rPickM*, if the constraint $C_1 = ((x = \lambda_{\mathrm{PS}}) \wedge idle(mpick_x) \wedge (x = 0)^{\uparrow})$, where $idle(mpick_x) = (x \leq \lambda_{\mathrm{PS}} \wedge x \leq 1)$ and $(x = 0)^{\uparrow}$ (i.e., $x \geq 0$), is satisfiable. Intuitively, C_1 denotes the constraint where λ_{PS} time units elapsed since clock x has started. In fact, C_1 is satisfiable (for example, with $\lambda_{\mathrm{PS}} = 0.5$ and $x = 0.5$). Therefore, it may evolve into the intermediate state $s_1' = (r_{good}, (x = \lambda_{\mathrm{PS}}) \wedge idle(mpick_x) \wedge (x = 0)^{\uparrow}, \lambda_{\mathrm{PS}}) = (r_{good}, (x = \lambda_{\mathrm{PS}}) \wedge x \leq 1, \lambda_{\mathrm{PS}})$. 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 $s_1 = (r_{good}, \lambda_{\mathrm{PS}} \leq 1, \lambda_{\mathrm{PS}})$.
- From the intermediate state s_0^x, the process may also evolve into the intermediate state s_2', by applying the rule *rPickA*, if the constraint $C_2 = ((x = 1) \wedge idle(mpick_x) \wedge (x = 0)^{\uparrow})$, where $idle(mpick_x) = (x \leq \lambda_{\mathrm{PS}} \wedge x \leq 1)$ 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 $s_2' = (r_{bad}, (x = 1) \wedge x \leq \lambda_{\mathrm{PS}}, 1)$. After clock pruning from intermediate state s_2', it becomes state $s_2 = (r_{bad}, \lambda_{\mathrm{PS}} \geq 1, 1)$.
- 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 s_1^x, the process may evolve into intermediate state s_3' by applying rule *rReply*, if the constraint $C_3 = ((x = 0) \wedge (\lambda_{\mathrm{PS}} \leq 1)^{\uparrow})$ is satisfiable, where $(\lambda_{\mathrm{PS}} \leq 1)^{\uparrow} = \lambda_{\mathrm{PS}} \leq 1$. The constraint is satisfiable and therefore the state evolves $s_3' = (Stop, \lambda_{\mathrm{PS}} \leq 1 \wedge (x = 0), \lambda_{\mathrm{PS}})$. After pruning of the non-active clock, it evolves into the terminal state $s_3 = (Stop, \lambda_{\mathrm{PS}} \leq 1, \lambda_{\mathrm{PS}})$. Since the terminal state is not caused by a bad activity, s_3 is considered as a good state, denoted by ✓ in Fig. 3.
- From state s_2, the process may also evolve into the terminal state $s_4 = (Stop, \lambda_{\mathrm{PS}} \geq 1, 1)$. Since the terminal state is caused by a bad activity, it is considered as a bad state, denoted by $\times$ in Fig. 3.

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

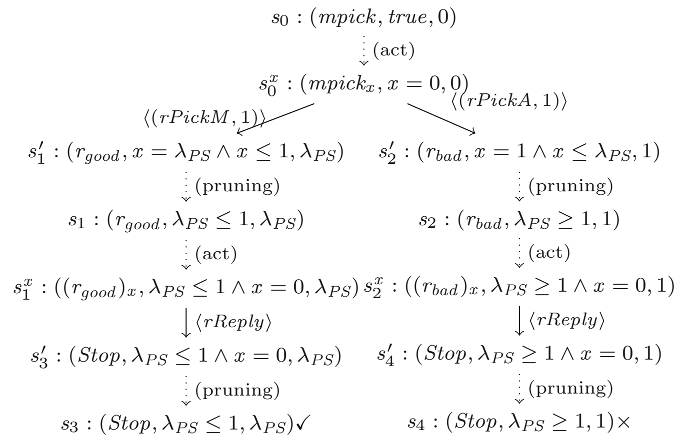


Fig. 3 Computing states of service CS (including intermediate states)

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. 3 to illustrate in detail the application of the semantics. The LTS of CS (without the intermediate states) is given in Fig. 4.

4.6 A technical result: the reachability condition

We defined the operational semantics of parametric composite service models as an LTS, the states of which contain information on clocks and parameters in the form of a constraint C. We now show that, for any reachable state of this LTS along a run, a parameter valuation π satisfies C iff the model valuated with π has an equivalent run. This is called the *reachability condition*. Similar results have been proved for parametric timed automata [41], parametric time Petri nets [76] or parametric stateful timed CSP [11].

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 $LTS_{\mathsf{CS}[\pi]}$. We will say that two runs are equivalent if they share the same discrete support, i.e., follow the same application of sequences of rules regardless of the actual timing values.

Definition 10 (*Equivalent runs*) Let CS be a parametric service model, and let π be a parameter valuation.

Let $\rho = \langle(v_0, P_0, C_0, D_0), seq_0, (v_1, P_1, C_1, D_1), \ldots, seq_{n-1}, (v_n, P_n, C_n, D_n)\rangle$ be a run of $LTS_{\mathsf{CS}[\pi]}$. Let $\rho' = \langle(v_0', P_0', C_0', D_0'), seq_0', (v_1', P_1', C_1', D_1'), \ldots, seq_{n-1}', (v_n', P_n', C_n', D_n')\rangle$ 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 $seq_i = seq_i'$ for $0 \leq i \leq n-1$.

The following lemma states that, given a run of $LTS_{\mathsf{CS}[\pi]}$, there exists a unique equivalent run in LTS_{CS}.

Proposition 1 *Let* CS *be a parametric service model, and let π be a parameter valuation. Let ρ_π be a run of $LTS_{\mathsf{CS}[\pi]}$. 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 $LTS_{\mathsf{CS}[\pi]}$, and given a state (v', P', C', D') of the equivalent run ρ' in LTS_{CS}, we show that these two runs are not only equivalent, but also that $C \subseteq C'$.

Base case From Definition 9, the initial state of LTS_{CS} is $(v_0, P_0, C_0, 0)$. The initial state of $LTS_{\mathsf{CS}[\pi]}$ 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 $LTS_{\mathsf{CS}[\pi]}$ 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 10, 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 a given sequence of rules seq 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 "Appendix A." 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:

$C_1 \subseteq C_1' \Longrightarrow C_1^{\uparrow} \subseteq C_1'^{\uparrow}$
$\Longrightarrow (x = \pi(\lambda_S) \wedge C_1^{\uparrow}) \subseteq (x = \lambda_S \wedge C_1'^{\uparrow})$
$\Longrightarrow C_2 \subseteq C_2'$.

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 $P_2 = P_2'[\pi]$. Variables are updated in the same manner on both sides, and hence, $v_2 = v_2'$. The proof is similar for other rules in "Appendix A."

Finally, 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 this sequence

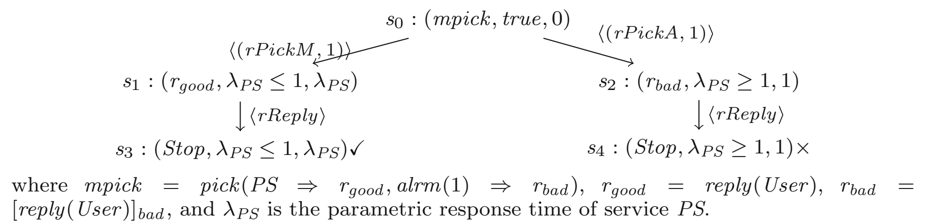


Fig. 4 LTS of service CS

of rules. Hence, there exists a unique run of LTS_{CS} equivalent to ρ_π and of length $m+1$. □

We now prove the dual result. Proposition 2 states that, given a run ρ of LTS_{CS}, there exists a unique equivalent run in $LTS_{\mathsf{CS}[\pi]}$, provided π satisfies the parametric constraint associated with the last state of ρ.

Proposition 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 $\pi \models C_n{\downarrow_\Lambda}$, there exists a unique run of $LTS_{\mathsf{CS}[\pi]}$ 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 $LTS_{\mathsf{CS}[\pi]}$, we show that these runs are not only equivalent, but also that $C = C'[\pi]$.

Base step: From Definition 9, the initial state of LTS_{CS} is $(v_0, P_0, C_0, 0)$. The initial state of $LTS_{\mathsf{CS}[\pi]}$ is $(v_0, P_0[\pi], C_0[\pi], 0)$. Since $C_0 = true$ 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 a sequence of rules *seq* in ρ. Let $\pi \models C'_2{\downarrow_\Lambda}$. Assume there exists a unique run of $LTS_{\mathsf{CS}[\pi]}$ equivalent to ρ and of length m, reaching state (v_1, P_1, C_1, D_1). From Definition 10, 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 "Appendix A." Recall that $C_1 = C'_1[\pi]$; since $P_1 = P'_1[\pi]$ (from Definition 10), 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 "Appendix A," we know that $C_2 = (x = \lambda_{\mathsf{S}} \wedge (C_1^{\uparrow}))$ and $C'_2 = (x = \lambda_{\mathsf{S}} \wedge (C_1'^{\uparrow}))$. Now, we have:

$$
\begin{aligned}
C_2 &= \big(x = \pi(\lambda_{\mathsf{S}}) \wedge (C_1^{\uparrow})\big) \\
&= \big(x = \pi(\lambda_{\mathsf{S}}) \wedge (C'_1[\pi])^{\uparrow}\big) \quad \text{(induction hypothesis)} \\
&= \Big(x = \pi(\lambda_{\mathsf{S}}) \wedge (C'_1 \wedge \bigwedge_{\lambda_i \in \Lambda} \lambda_i = \pi_i)^{\uparrow}\Big) \quad \text{(definition of valuation)} \\
&= \big(x = \pi(\lambda_{\mathsf{S}}) \wedge (C'_1)^{\uparrow}\big) \wedge \bigwedge_{\lambda_i \in \Lambda} \lambda_i = \pi_i \quad \text{(property of time elapsing)} \\
&= \big(x = \lambda_{\mathsf{S}} \wedge (C'_1)^{\uparrow}\big) \wedge \bigwedge_{\lambda_i \in \Lambda} \lambda_i = \pi_i \quad \text{(definition of valuation)} \\
&= C'_2 \wedge \bigwedge_{\lambda_i \in \Lambda} \lambda_i = \pi_i \quad \text{(definition of } C'_2\text{)} \\
&= C'_2[\pi] \quad \text{(definition of valuation)}
\end{aligned}
$$

Note that adding $x = \lambda_{\mathsf{S}}$ while keeping satisfiability of the expression is only true because $\pi \models C'_2{\downarrow_\Lambda}$. 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, and hence, $v_2 = v'_2$. The proof is similar for other rules in "Appendix A."

The proof of uniqueness is identical to that of Proposition 1. □

Propositions 1 and 2 give the following theorem.

Theorem 1 (Reachability condition) *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).*

There exists a run of $LTS_{\mathsf{CS}[\pi]}$ equivalent to ρ iff $\pi \models C_n{\downarrow_\Lambda}$.

5 Synthesizing the static LTC

Given $\mathsf{CS} = (\mathcal{V}, \Lambda, P_0, C_0)$, 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) $C_L \in \mathcal{C}_\Lambda$, 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. The sLTC will be given in the form of an NNCC over Λ. 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

Let $\lambda_i \in \mathbb{Q}_{\geq 0}$ be the parametric response time of component service S_i for $i \in \{1, \ldots, n\}$, and let $\Lambda = \{\lambda_1, \ldots, \lambda_n\}$ be the set of component service parametric response times.

Using constraints over Λ, 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 Λ. An example of a local time requirement is $(\lambda_1 \leq 6) \wedge (\lambda_2 \leq 5)$. 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., $(\lambda_2 \leq \lambda_1 \Rightarrow \lambda_1+\lambda_2 \leq 6) \wedge (\lambda_1 \leq \lambda_2 \Rightarrow \lambda_1 \leq 6)$.

In the following, we will propose a technique to synthesize the static LTC in the form of a convex over Λ. We first give an intuition concerning how to handle the good states (Sect. 5.2) and the bad states (Sect. 5.3); then, we give the full synthesis algorithm (Sect. 5.4), apply it to an example (Sect. 5.5) and prove its soundness (Sect. 5.6).

5.2 Addressing the good states

We assume a composite service CS and its LTS $LTS_{\mathsf{CS}} = (S, s_0, \mathsf{Sequences(Rules)}, \delta)$; let S_{good} be the set of all good states of LTS_{CS}. We make two observations here. First, from Theorem 1, a good state $s_g = (v_g, P_g, C_g, D_g) \in S_{good}$ 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 $s_g \in S_{good}$, we require the constraint $(C_g{\downarrow_\Lambda} \Rightarrow (D_g \leq T_G))$ 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 must include the conjunction of such constraints for each good state $s_g \in S_{good}$, that is:

$$\bigwedge_{(v_g,P_g,C_g,D_g)\in S_{good}} (C_g{\downarrow_\Lambda} \Rightarrow (D_g \leq T_G)).$$

Example 8 Let us consider a composite service CS whose process component is $P_0 = pick(\mathsf{S} \Rightarrow sInv(\mathsf{S}_1), alrm(1) \Rightarrow sInv(\mathsf{S}_2))$, where S is a component service. Assume that $sInv(\mathsf{S}_j)$ is a component service with parametric response time λ_j, for $j \in \{1, 2\}$, and S has a response time λ_S. Suppose the global time requirement of the composite service CS is to respond within 5 s. Fig. 5 shows the LTS of CS.

For composite service CS in Fig. 5, we have two good states (states s_3 and s_4), and the synthesized local time requirement for composite service CS is:

$$(\lambda_\mathsf{S} \leq 1) \Rightarrow (\lambda_\mathsf{S} + \lambda_1 \leq 5) \wedge (\lambda_\mathsf{S} \geq 1) \Rightarrow (1 + \lambda_2 \leq 5)$$

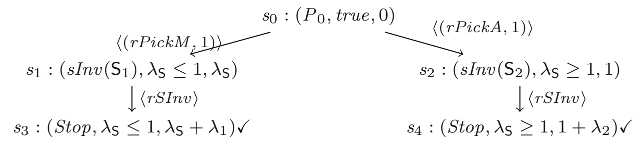


Fig. 5 LTS of composite service CS

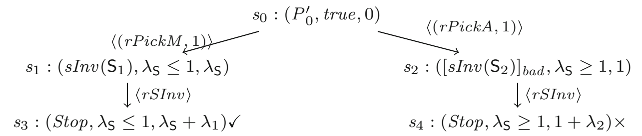


Fig. 6 LTS of composite service CS′

5.3 Addressing the bad states

Another goal we want to achieve is to avoid all bad states in LTS_{CS}. Let S_{bad} be the set of all bad states of service LTS_{CS}. Given a bad state $s_b = (v_b, P_b, C_b, D_b) \in S_{bad}$, 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 Λ, i.e., we require that $\neg(C_b){\downarrow_\Lambda}$ be satisfiable because of the reachability condition (Theorem 1). In addition to the good state constraint given in Sect. 5.2, the synthesized sLTC for CS must also include the conjunction of such constraints for each bad state $s_b \in S_{bad}$, that is:

$$\bigwedge_{(v_b,P_b,C_b,D_b)\in S_{bad}} (\neg(C_b){\downarrow_\Lambda}).$$

Example 9 Consider a variant CS′ of Example 8, where $sInv(\mathsf{S}_2)$ is now treated as a bad activity, denoted by $[sInv(\mathsf{S}_2)]_{bad}$. This service results in the LTS shown in Fig. 6, where state s_4 is a bad state. From Theorem 1, a way to avoid the reachability of s_4 is to negate its associated constraint C. Therefore, the local time requirement for composite service CS′ is $(s_3.C{\downarrow_\Lambda} \Rightarrow (s_3.D \leq T_G)) \wedge \neg\ (s_4.C{\downarrow_\Lambda})$: the first term guarantees the reachability of s_3 while the second term guarantees the non-reachability of s_4. Therefore, this NNCC ensures that any complete run of the service ends in a good state. (This will be proved in Sect. 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 of CS. Algorithm 1 simply calls $\mathsf{synthRec}(s)$ applied to the initial state s_0; this latter algorithm synthRec is given in Algorithm 2.

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

Algorithm 1: synthSLTC(CS)

input : Composite service model CS with LTS LTS_{CS} of initial state s_0
output: The sLTC $C_L \in \mathcal{NC}_\Lambda$

1 **return** synthRec(s_0);

Algorithm 2: synthRec(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** $(s.C{\downarrow_\Lambda} \Rightarrow (s.D \leq T_G))$;
3
4 **else if** *s is a bad state* **then**
5 | **return** $\neg\ (s.C{\downarrow_\Lambda})$;
6 **else**
| `// s is a non-terminal state`
7 | **return** $\bigwedge_{s' \in succ(s)}$ synthRec(s') ;

$s.C{\downarrow_\Lambda} \Rightarrow (s.D \leq T_G)$ (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 $s.C{\downarrow_\Lambda}$ is returned (line 4). Finally, 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).

5.5 Application to the running example

Consider again the running example SMIS introduced in Sect. 2. Assume the parametric response times of FS, PS and DS are λ_{FS}, λ_{PS} and λ_{DS}, respectively. Recall that $T_G = 3$.

Figure 7 shows the LTS of SMIS. The sLTC resulting from the application of synthSLTC is:

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

After simplification[3] using Z3 [29], a state-of-the-art satisfiability modulo theories (SMT) solver developed by Microsoft Research, we get the following sLTC:

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

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 s, and component services DS and FS where the summation of their response times should be less than or equal to 3 s. For example, a suitable valuation is π such that $\pi(\lambda_{\mathrm{FS}}) = 0.5$, $\pi(\lambda_{\mathrm{DS}}) = 1.5$ and $\pi(\lambda_{\mathrm{FS}}) = 0.8$.

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

5.6 Termination and soundness of synthSLTC

5.6.1 Termination

Lemma 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 3 *Let* CS *be a service model. Then* synthSLTC(CS) *terminates.*

Proof From Lemma 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 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. □

Remark 1 (*Complexity of Algorithm* 2) First, note that all states of LTS_{CS} are explored by Algorithm 2: indeed, the algorithm is recursively called on non-terminal states and stops only on terminal states—that have no successors anyway. So, the algorithm time is constant in the number of states of LTS_{CS}. In addition, the number of conjuncts in the result of Algorithm 2 is at most the number of states of LTS_{CS}, and less if not all states are terminal states.

5.6.2 Soundness

Let us 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.

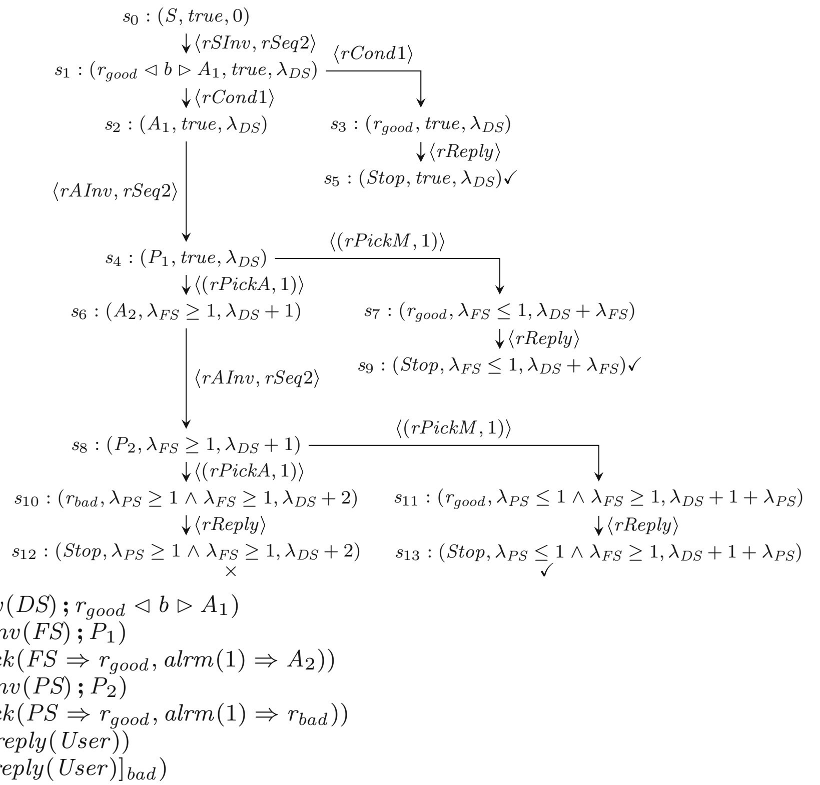
In the following, given a run ρ_π of $LTS_{\mathsf{CS}[\pi]}$, from Proposition 1 we can safely refer to the run of LTS_{CS} equivalent to ρ_π.

The following lemmas will be used to prove the subsequent Theorem 2.

Lemma 2 *Let* CS *be a service model. Let* $\pi \models$ synthSLTC(CS). *Then no bad state is reachable in $LTS_{\mathsf{CS}[\pi]}$.*

Proof Let $K =$ 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

Fig. 7 LTS of the SMIS



constraints of all bad states. Hence, from Theorem 1, the bad states are unreachable for any $\pi \models K$. □

Lemma 3 *Let* CS *be a service model. Let* $\pi \models$ synthSLTC(CS). *Then any complete run of* $LTS_{\mathsf{CS}[\pi]}$ *ends in a good state.*

Proof First, note that the initial state s_0 is reachable in $LTS_{\mathsf{CS}[\pi]}$ (since $s_0.C = true$).

If the initial state is the only state, then from Lemma 2, it is also not a bad state; hence, it is a good state. Now, if it is not the only state, from the fact that all runs of LTS_{CS} end either in a good state or in a bad state, from the absence of bad states (Lemma 2), and from Theorem 1, then any run of $LTS_{\mathsf{CS}[\pi]}$ ends in a good state. □

Lemma 4 *Let* CS *be a service model. Let* $\pi \models$ synthSLTC(CS). *Then for all good state* (v, P_g, C, d) *of* $LTS_{\mathsf{CS}[\pi]}$, $d \leq T_G$.

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 9, C is satisfiable (and hence $C{\downarrow_\Lambda}$ too). Since s_g is a good state, Algorithm synthRec added a constraint $C{\downarrow_\Lambda} \Rightarrow D \leq T_G$ to the result. Hence, synthSLTC(CS) $\subseteq (C{\downarrow_\Lambda} \Rightarrow D \leq T_G)$. Now, for any $\pi \models$ synthSLTC(CS), we have that $\pi \models (C{\downarrow_\Lambda} \Rightarrow D \leq T_G)$, and hence, all reachable states in $LTS_{\mathsf{CS}[\pi]}$ are such that $d \leq T_G$. □

We can now formally state the soundness of synthSLTC.

Theorem 2 *Let* CS *be a service model. Let* $\pi \models$ synthSLTC(CS). *Then:*

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

Proof From Lemmas 3 and 4. □

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 $\pi \models$ synthSLTC(CS) necessarily satisfies (at least) one of the good states' constraints, i.e., $\pi \models s_i.C{\downarrow_\Lambda}$ for some $s_i \in S_g$.

Indeed, recall synthSLTC(CS) is a conjunction of good and bad constraints. In the following proposition, we show that the good constraints of the form $(C_1 \Rightarrow r_1 \wedge \cdots \wedge C_n \Rightarrow r_n)$ will not hold trivially by just having $C_i = false$, for all $i \in \{1, \ldots, n\}$.

Proposition 4 *Let* CS *be a service model, and* S_{good} *be the set of all good states in* LTS_{CS}. *Let* $\pi \models$ synthSLTC(CS). *Then* $\exists s \in S_{good} : \pi \models s.C{\downarrow_\Lambda}$.

Proof From Algorithm 2, synthSLTC(CS) is a conjunction of "good" constraints (accumulated at line 2) and "bad" constraints (accumulated at line 4). That is, assume synthSLTC(CS) $= (C_g \wedge C_b)$, where $C_g = \bigwedge_{s_i \in S_{good}} (s_i.C{\downarrow_\Lambda} \Rightarrow (s_i.D \leq T_G))$, and T_G be the global time constraint, and $C_b = \bigwedge_{s_j \in S_{bad}} \neg(s.C_j{\downarrow_\Lambda})$. Hence, since $\pi \models$ synthSLTC(CS) then $\pi \models C_g$ and hence $\exists s \in S_{good} : \pi \models s.C{\downarrow_\Lambda}$. □

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 $A \lhd a = 1 \rhd B$, 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 Sect. 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 Sect. 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 might 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-valued constants; this makes the synthesized LTC contain less uncertainty and be more precise.

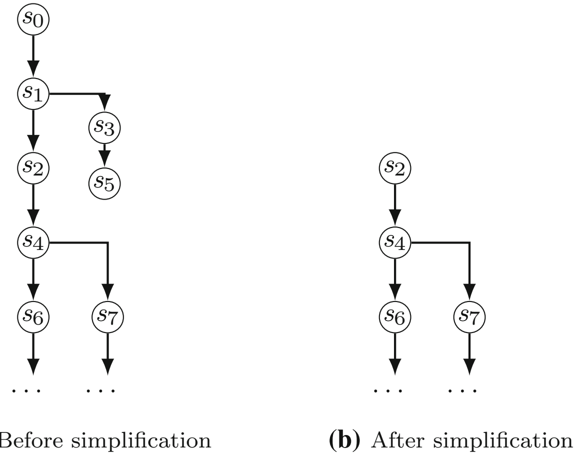


Fig. 8 LTS simplification of SMIS

Example 10 For example, consider the SMIS composite service, the LTS of which is depicted in Fig. 7. Assume a valuation π satisfying the *sLTC*. 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 time units, 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 response time $\pi(\lambda_{DS})$, 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. 7), as well as the successors of these past states that were not met in practice (e.g., s_3 and s_5 in Fig. 7), because another branch was taken at runtime. We show the LTS of SMIS before and after simplification in Fig. 8a, b, respectively.

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

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

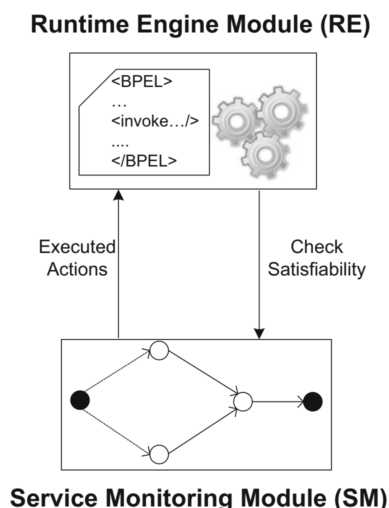


Fig. 9 Service adaptation framework

6.2 Runtime adaptation of a BPEL process

We now introduce a service adaptation framework to improve the conformance of global time requirement for a composite service. The architecture of the framework is shown in Fig. 9. There are two modules in the framework—the runtime engine module (RE) and the service monitoring module (SM). RE provides an environment for the execution of a BPEL service; here, we use Apache ODE [35], an open-source BPEL engine. We instrument the runtime component of Apache ODE to communicate with 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 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 sequences of rules from RE are used to update the active state $s_a \in S$ of LTS stored in SM. The sequence of rules is also 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).

Example 11 Consider again the running example SMIS in Sect. 2. An example of S and S_{bak}, are services FS and PS, respectively.

In the following, we introduce the details on the synthesis of rLTC (Sect. 6.3) and satisfiability checking (Sect. 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: $\bigsqcap((cons_1.g, cons_1.b), \ldots, (cons_n.g, cons_n.b))$ denotes $((cons_n.g \wedge \cdots \wedge cons_n.g), (cons_n.b \wedge \cdots \wedge cons_n.b))$.

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 $g, b \in \mathcal{C}_\Lambda$. In this pair, g (resp. b) denotes the constraint associated with 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 particular, given a state s, free variables d_f and r_f in $s.rLTC$ are to be substituted by the delay component $s.D \in \mathcal{L}_\Lambda$ and the actual delay $r \in \mathbb{R}_{\geq 0}$ from the initial state to the state s, respectively.

Let us now explain synthRLTC in detail. 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. 7). At runtime, assume the active state is at state s_{13}, and assume that it takes $r \in \mathbb{R}_{\geq 0}$ time units 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., $t_{\mathsf{DS}} + 1 + t_{\mathsf{PS}}$, 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

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

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 (line 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 9).

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 consist of invoking a faster backup service S'_i instead of S_i.

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

```
input  : LTS of the parametric composite service CS, Active state
         s_a ∈ S, Elapsed time r ∈ ℝ≥0, Set of parametric response
         times Λ, Parameter valuation π
output : True if the local time constraint at s_a is satisfiable, false
         otherwise
1 return Is_Sat((⋀_{1≤i≤n} λ_i ≤ π(λ_i)) ⇒ (s_a.rLTC ∧ (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 ($\bigwedge_{1\leq i\leq n} \lambda_i \leq \pi(\lambda_i)$), 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* returns true iff the input constraint is satisfiable.

6.5 Termination and soundness of synthRLTC

6.5.1 Termination

Proposition 5 *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 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 3 formally states the correctness of our runtime refinement algorithm.

Theorem 3 *Let* CS *be a service model. 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 $\pi \models$ synthRLTC(CS, LTS_{CS}, s). *Then:*

1. *there exists a run in* $LTS_{CS[\pi]}$ *ending in some state* s_π *such that this run is equivalent to a run of* LTS_{CS} *ending in 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 Proposition 2.
2. From Definition 5, 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 3, 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 4, for all good state of $LTS_{CS[\pi]}$, $d \leq T_G$, which gives the result. □

Remark 2 (*Complexity of Algorithm 3*) First, note that all states of LTS_{CS} are explored by Algorithm 3: indeed, the algorithm is recursively called on non-terminal states and stops only on terminal states—that have no successors anyway. So, the algorithm time is constant in the number of states of the sub-LTS of $LTS_{CS[\pi]}$ starting from s_π.

Let us now investigate the worst-case number of conjuncts in the result of Algorithm 3. The algorithm returns the good conjuncts ($cons.g$), the bad conjuncts ($cons.b$) and a last term ("$d_f = s.D$") (line 10 in Algorithm 3). Any good terminal state or bad terminal state adds exactly one conjunct to either $cons.g$ or $cons.b$. Therefore, the number of conjuncts is exactly the number of terminal states, plus one due to the last term.

6.6 Discussion

Termination From Proposition 5, our method terminates due to the fact that BPEL composite services do not support recursion, and Assumption 1 on the loop activities ensuring that the upper bound on the number of iterations and the time of execution are known. We briefly 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., [31]), or could be provided by the user otherwise. In the worst case, an alternative option is 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., [58].

7 Evaluation

As a proof of concept, we applied our method to several examples. After briefly presenting our implementation, we describe the examples we use (Sect. 7.1). We then evaluate our methods for the synthesis of local time requirement at the design time (Sect. 7.2) and for the runtime refinement (Sect. 7.3).

Implementation We implemented our algorithms for synthesizing the sLTC and rLTC in SELAMAT, a tool developed in C♯. We use an *ad-hoc* input syntax very close to that of Definition 1. Our prototype implementation uses basic state space reduction techniques, notably zone inclusion (see, e.g., [40,59] for recent such techniques in the (parametric) timed setting), to prune whole branches of the state space. The front-end GUI relies on the PAT model checker [69].

The simplification of the final results of sLTC and rLTC is achieved using Microsoft Z3 [29]. For the runtime adaptation, we use 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. The tool and examples can be downloaded at [71].

7.1 Examples

7.1.1 Stock Market Indices Service (SMIS)

This is the running example introduced in Sect. 2.

7.1.2 Computer purchasing services (CPS)

The goal of a CPS 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 3 s. The CPS workflow is shown in Fig. 10. 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 reply message from LS and BS, the result of the computer purchasing will be returned to the user.

7.1.3 Travel booking service (TBS)

The goal of a travel booking service (TBS) is to provide a combined flight and hotel booking service by integrating

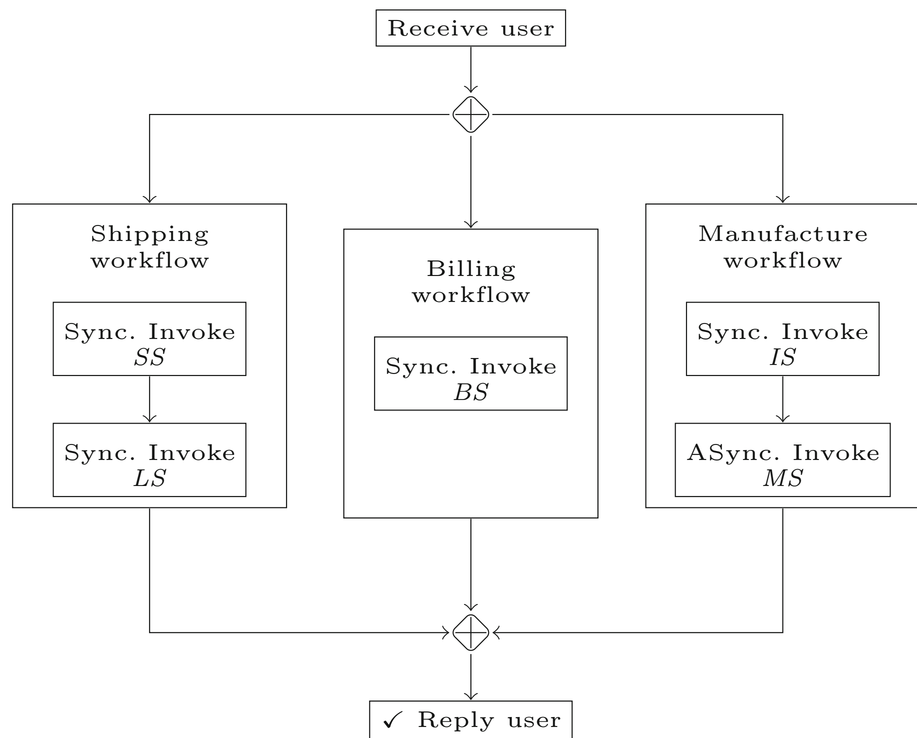


Fig. 10 Computer purchasing service (CPS)

two independent existing services. TBS provides an SLA for its subscribed users, saying that it must respond within 5 s 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. 11. 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 2 s, then FS is abandoned, and another backup flight service FS_{bak} is invoked. If FS_{bak} returns within 1 s, 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.

7.1.4 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 service (WS) and distance service (DS). The global requirement of the RS is to respond within 5 s. The RS workflow is given in Fig. 12. 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.

7.2 Synthesis of local time requirement

7.2.1 Environment of the experiments

We run our algorithms to synthesize the sLTC and rLTC for the four examples on a computer with Intel Core I5 2410M CPU with 4 GB RAM.

7.2.2 Evaluation results

The details of the synthesis are shown in Table 1. 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

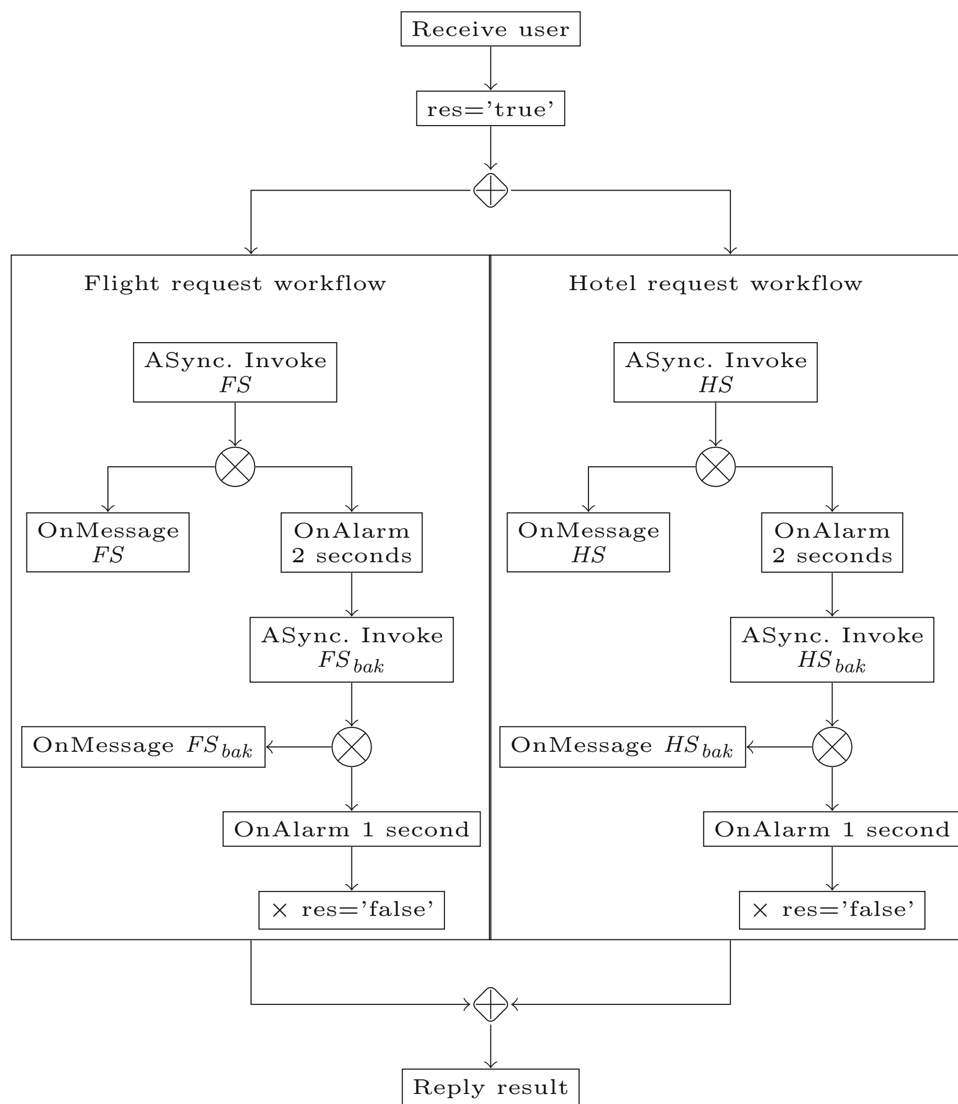


Fig. 11 Travel booking service (TBS)

the average time (in seconds) spent for synthesizing sLTC (for the entire LTS), and rLTC (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 (less than 2 s) for TBS is considered to be reasonable.

The synthesized sLTC for SMIS is given in Sect. 5.5, while the synthesized sLTC for CPS, TBS and RS is shown in Fig. 13. Note that λ_{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 is used for runtime adaptation at runtime. We will evaluate the runtime adaptation of a composite service with rLTC in the following section.

7.3 Runtime adaptation

We now conduct experiments to answer the following two questions:

Q1 What is the *overhead* of the runtime adaptation?
Q2 What is the *improvement* provided by the runtime adaptation?

7.3.1 Environment of the experiments

The evaluation was conducted using two different physical machines, connected by a 100 Mbit LAN. One machine is running Apache ODE [35] to host the RE module to execute the BPEL program, configured with Intel Core I5 2410M CPU with 4 GB RAM. The other machine hosts the SM module, configured with Intel I7 3520M CPU with 8 GB RAM.

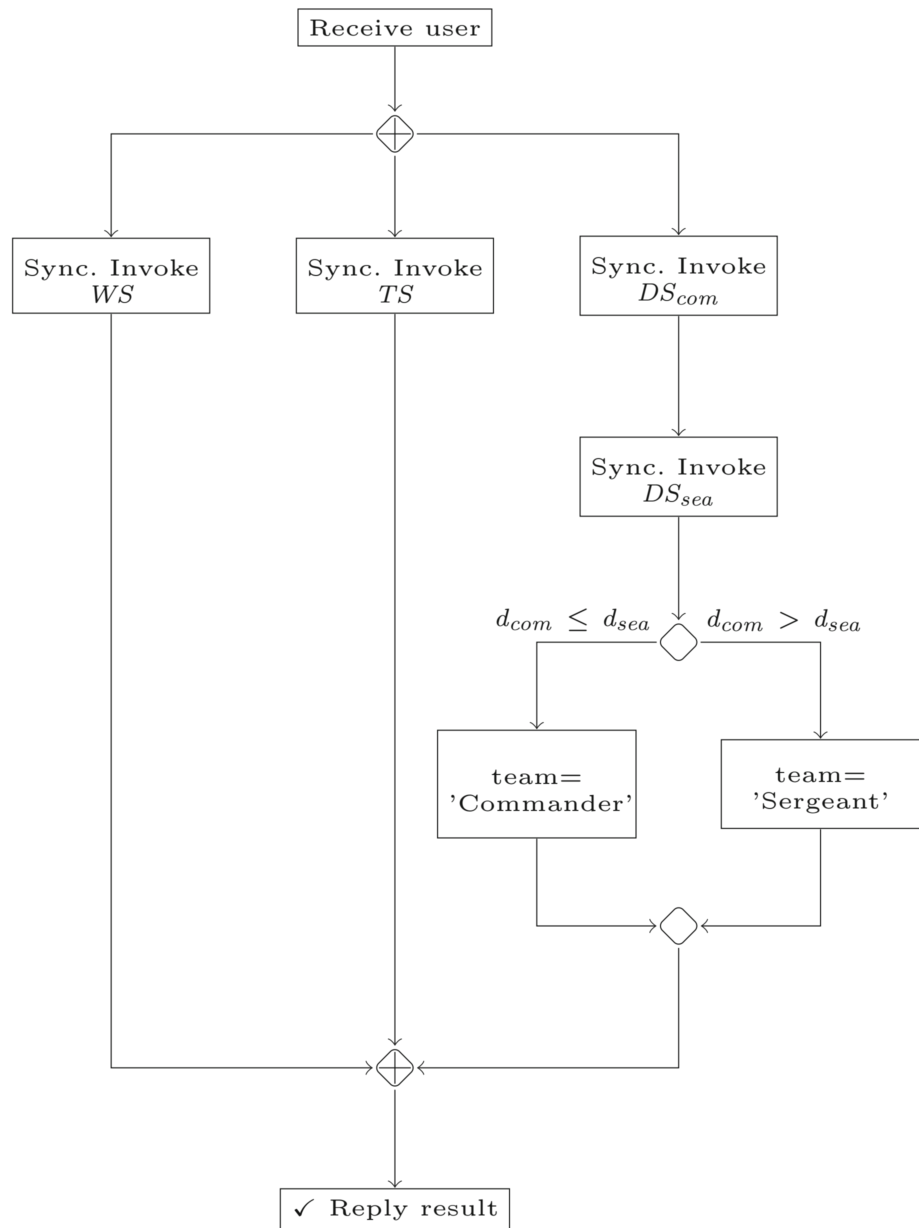


Fig. 12 Rescue team service (RS)

Table 1 Synthesis of sLTC and rLTC

Example	#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

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 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 model, where a component service S_i of CS has a stipulated response time $\pi(\lambda_i) \in \mathbb{Q}_{\geq 0}$. Then $R(\mathsf{S}_i)$ will be assigned with a response time within the stipulated response time $\pi(\lambda_i)$ with a probability of $p_c \in \mathbb{Q}_{\geq 0} \cap [0, 1]$. p_c is the *response time conformance threshold*. More specifically, $R(\mathsf{S}_i)$ will be assigned with a value in $[0, \pi(\lambda_i)]$ uniformly with a probability of p_c and assigned to a value in $(\pi(\lambda_i), \pi(\lambda_i) + t_e]$ uniformly with a

Fig. 13 Synthesized sLTC

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

(a) sLTC for CPS

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

(b) sLTC for RS

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

(c) sLTC for TBS

Table 2 Satisfiability checking

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

probability of $1 - p_c$. $t_e \in \mathbb{R}_{\geq 0}$ is the *exceeding threshold*; and assume after $\pi(\lambda_i) + t_e$ 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 $\rho(\mathsf{CS}, \mathsf{AM}, E)$, where the first argument is the composite service CS that is running, the second argument $\mathsf{AM} \in \{rr, \varnothing\}$ is the adaptive mechanism where rr denotes the runtime adaptation, and $\varnothing$ denotes no runtime adaptation.

7.3.2 Evaluation results

We conducted two experiments Exp1 and Exp2, to answer the questions Q1 and Q2, respectively. Each experiment goes through 10,000 rounds of simulations, 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 $\pi(\lambda_i)$, there exists a backup service S'_i, with a stipulated response time $\pi(\lambda_i)/2$ 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, the faster backup service S'_i will be invoked instead, as a mitigation procedure.

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 $\rho(\mathsf{CS}, rr, E)$ and non-adaptive run $\rho(\mathsf{CS}, \varnothing, E)$. We have modified the runtime adaptation mechanism for rr so 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 $\rho(\mathsf{CS}, rr, E)$ and $\rho(\mathsf{CS}, \varnothing, E)$ invoke the same set of component services, so that we can effectively compare the overhead of $\rho(\mathsf{CS}, rr, E)$.

Results Suppose at round k, the times spent for $\rho(\mathsf{CS}, rr, E)$ and $\rho(\mathsf{CS}, \varnothing, E)$ 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 Eq. (1).

$$\text{Avg. overhead} = \left(\sum_{i=1}^{k} O_i\right)/k \tag{1}$$

The main source of overhead for runtime adaptation comes from the satisfiability checking with Algorithm 4. We make use of Z3 [29] for this purpose. Other sources of overhead include update of active state in SM, and communications between SM and RE.

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 2 shows the results of satisfiability checking. The average number of satisfiability checking for each round (Avg. #SAT) is calculated using Eq. (2) where N_i is the total number of satisfiability checking for ith 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 Eq. (3), where T_i is the time spent on satisfiability checking for ith round. Table 2 shows that the satisfiability checking has contributed most of the overhead of runtime adaptation.

$$\text{Avg. \#SAT} = \left(\sum_{i=1}^{r} N_i\right)/r \tag{2}$$

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

Experiment Exp2 In this second experiment, we measure the improvement for the conformance of global constraints due

Table 3 Improvement of runtime conformance

	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	10, 000	9743	2.64	0.384
	0.8	10, 000	9364	6.79	0.779
	0.7	10, 000	8460	18.20	0.948
	0.6	10, 000	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

to rr. Given a composite service CS, an execution configuration E, two runs $\rho(\mathsf{CS}, rr, E)$ and $\rho(\mathsf{CS}, \varnothing, E)$ 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 Eq. (4).

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

Results The experiment results can be found in Table 3. The Improvement (%) column provides the information of improvement (in percentage) that is calculated using Eq. (4). The *Avg. backup service* column provides the average number of backup service used (calculated by summing up 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.

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.

7.4 Threats to validity

Our experiments show a good efficiency of our technique for the examples we considered; these are arguably on the smaller side, but we claim that they are non-trivial enough to not be analyzable by hand, and therefore, our technique proposes what we believe to be a valuable contribution.

8 Related work

Model-based analysis of Web services using LTSs Our method is related to using LTSs for model-based analysis of Web services. In [18], 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. [33,34] transform BPEL process into FSP [51], subsequently using a tool named "WS-Engineer" for checking safety and liveness properties. Simmonds et al. [66] propose a user-guided recovery framework for Web services based on LTSs. Our work uses LTSs in synthesizing local time requirement.

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) [5] and parametric time Petri nets (PTPNs) [76] for solving such problems has received recent attention. In particular, in [26,36,46], parametric constraints are inferred, guaranteeing the feasibility of a schedule using PTAs extended with stopwatches (see, e.g., [1]). In [11], 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., [12]). 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.

Finding suitable quality of service Our method is related to the finding of a suitable quality of service (QoS) for the system [77]. The authors of [77] 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 [14] 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 [3]. 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 calls, without considering complex control flows and timing requirements.

Response time estimation Our approach is also related to response time estimation. In [44], 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. [53] 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 work [48] complements with this work by proposing a method on building LTCs that under-approximate the sLTC of a composite service. The under-approximated LTCs consist 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. [58] 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. [16] propose an idea of self-supervising BPEL processes by supporting both service monitoring and recovery for BPEL processes. They propose using Web service constraint language (WSCoL) to specify the monitoring directives to indicate properties that need to 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 our monitoring approach, as discussed in Sect. 6.2. Our previous work [73] 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.

Service selection In [78,79], the authors present an approach that makes use of global planning to search dynamically for the best services component for service composition. Their approach involves the use of mixed integer programming (MIP) techniques to find the optimal selection of component services. Ardagna et al. [13] extend the MIP methods to include local constraints. Cardellini et al. [20] propose a methodology to integrate different adaptation mechanisms for combining concrete services to an abstract service, in order to achieve a greater flexibility in facing different operating environments. Our work is orthogonal to aforementioned works, as it does not assume particular formulation of the MIP problems.

Although the method in aforementioned works efficiently for small case studies, it suffers from scalability problems when the size of the case studies becomes larger, since the time required grows exponentially with the size of problem. To address this problem, Yu et al. [77] propose a heuristic algorithm that could be used to find a near-optimal solution. The authors proposed two QoS compositional models, a combinatorial model and a graph model. The time complexity for the combinatorial model is polynomial, while the time complexity for the graph model is exponential. However, the algorithm does not scale with the increasing number of Web services. To address this problem, Alrifai et al. present an approach that pruned the search space using skyline methods, and they make use of a hierarchical k-means clustering method [49] for representative selection. The work of Alrifai

et al. is the closest to ours. Despite its reasonable performance, a limitation for the MIP approach is that it cannot deal with nonlinear objective functions or aggregated constraints. To address this problem, Canfora et al. [19] have formulated the problem as a genetic algorithm problem. Genetic algorithms (GA) are algorithms based on stochastic search methods that support nonlinear objective functions. Two different GAs encodings are proposed in [19,80]. In [80], the authors propose to encode the chromosome using binary strings, and every gene is a chromosome representing a service candidate with value 0 (respectively, 1) that represents the unselected (respectively, selected) service. Therefore, the length of the genome can be very long, given a large number of service candidates. In [19], the authors propose to encode the chromosome using an integer value which represents the index of the concrete services stored in an array. This coding scheme results in shorter chromosomes, and the length of a chromosome is independent of the number of service candidates. In [37], Gao et al. propose a tree coding scheme to represent the service composition. They reported a 40% performance improvement with respect to the single-dimension coding scheme used in [19]. This is because the tree coding scheme does not need to recalculate the entire fitness value each time compare to the single-dimension encoding. Our work does not assume any particular encoding scheme and it can be used with any existing coding techniques. In [2], Ai et al. proposed an approach extending the GA methods for handling interservice dependencies and conflicts using a penalty-based genetic algorithm. Our work does not assume a particular fitness function. In [50], Ma et al. proposed an enhanced initial population policy and an evolution policy based on population diversity and a relation matrix coding scheme. They considered all concrete services for each service class starting from the initial population. Different from their approach, we only consider a subset of services with high local utility value from the start, and we progressively add more services later on. In [70] the problem of functionally equivalent service composition is considered.

Verification of services Concerning verification of services, Filieri et al. [32] 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 [23,24] 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. [64] 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, at runtime, [64] 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 at runtime, and this allows a more efficient runtime analysis.

BPEL In [38] a template-based system is used to reconfigure service composition, using BPEL. In [61], service composition using RESTful (Representational State Transfer) is performed using the BPEL extension "BPEL for REST." In [30], a tool based on Services Creation Environments and BPEL is proposed that also allows translation to Java.

The work [75] automates the formalization and verification process of BPEL. It extends the existing spring framework to represent BPEL activities with Java bean, which is subsequently transformed into XML-based formal model (like colored Petri nets) for verification. The work [55] also transforms BPEL services into probabilistic labeled transition systems, which are then used to conduct probabilistic model checking to verify reliability properties on the BPEL models. Their word did not consider timing requirements.

Surveys Finally, composition of Web services has been recently surveyed. In [60] the larger domain of composition of convergent services is surveyed; however, BPEL is still surveyed in this work. In [47], a taxonomy of Web service composition is provided with different directions surveyed such as language, knowledge reuse, automation, tool support, execution platform or target user.

9 Conclusion and future work

9.1 Conclusion

We have presented a model-based approach for synthesizing local time constraints for component services of a composite service CS, knowing its global time requirement. Our approach makes use of parameterized timed techniques.

We first proposed a design time 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, we use 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 ser-

vice. The rLTC is then used to validate whether the composite service can still satisfy the global time requirement at runtime.

As a proof of concept, we have implemented our approach into a tool SELAMAT and applied it to four service composition examples. Our experiments show that the runtime refinement leads to an improvement of the global time requirement, with limited overhead.

9.2 Future work

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

General and dedicated optimizations First, the goal of our work is to propose a full framework for analyzing composition of Web services using parametric timings; therefore, integrating existing state space reduction techniques is perhaps a more practical work, orthogonal to the original goal of our approach. Nevertheless, in order to address huge sets of services, one could use efficient state-of-the-art techniques developed for timed systems or parametric timed systems such as (parametric) data difference bound matrices [41,62], efficient L/U-zone abstractions [40,59], convex state merging [9], integer-hull abstractions [10,43] or abstraction-refinement algorithms [63].

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

Constraints satisfiability Regarding our implementation, the bottleneck seems to be the satisfiability test using Z3; from our experience, switching to a polyhedra library (such as the Parma Polyhedra Library [15]) may give better results and could help render our work scalable.

Uncertain response times 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., [22,42]). An even more challenging problem 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.

Statistical model checking Finally, when concurrent systems with or without timing constraints are too huge to be analyzed in an exact manner, a recent trend is to propose non-exact techniques, and notably statistical model checking. This technique could be used for compositions of Web services arguably too large to be analyzed in an exact manner. Recent techniques developed in the timed setting (e.g., [27,45,52]) could be applied to our formalism.

A Operational semantics

Set of rules for the transition relation $\hookrightarrow$

Let $mpick = pick(\biguplus_{i=1}^{n} \mathsf{S}_i \Rightarrow P_i, \biguplus_{j=1}^{k} alrm(a_j) \Rightarrow Q_j)$

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

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

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

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

$$\frac{}{(v, mpick_x, C, D) \overset{\langle (rPickM, i) \rangle}{\hookrightarrow} (v', P_i, (x = \lambda_i) \wedge\ idle(mpick_x) \wedge C^{\uparrow}, D + \lambda_i)} [\ rPickM\ ]$$

$$\frac{}{(v, mpick_x, C, D) \overset{\langle (rPickA, j) \rangle}{\hookrightarrow} (v', Q_j, (x = a_j) \wedge\ idle(mpick_x) \wedge C^{\uparrow}, D + a_j)} [\ rPickA\ ]$$

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

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

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

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

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

$$\frac{(v, A, C, D) \overset{seq}{\hookrightarrow} (v', Stop, C', D')}{(v, A \,;\, B, C, D) \overset{seq+\langle rSeq2 \rangle}{\hookrightarrow} (v', B, C', D')} [\, rSeq2 \,]$$

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

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

References

1. Adbeddaïm, Y., Maler, O.: Preemptive job-shop scheduling using stopwatch automata. In: Katoen, J.P., Stevens, P. (eds.) TACAS, Lecture Notes in Computer Science, vol. 2280, pp. 113–126. Springer, Berlin (2002). https://doi.org/10.1007/3-540-46002-0_9
2. Ai, L., Tang, M.: A Penalty-Based Genetic Algorithm for QoS-Aware Web Service Composition with Inter-service Dependencies and Conflicts, pp. 738–743. IEEE Computer Society, Washington (2008). https://doi.org/10.1109/CIMCA.2008.104
3. Alrifai, M., Risse, T.: Combining global optimization with local selection for efficient QoS-aware service composition. In: Quemada, J., León, G., Maarek, Y.S., Nejdl, W. (eds.) WWW, pp. 881–890. ACM, New york (2009). https://doi.org/10.1145/1526709.1526828
4. Alur, R., Dill, D.L.: A theory of timed automata. TC **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
5. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Kosaraju, S.R., Johnson, D.S., Aggarwal, A. (eds.) STOC, pp. 592–601. ACM, New York (1993). https://doi.org/10.1145/167088.167242
6. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., IBM, Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version, version 2.0 (2007)
7. Amazon: Amazon elastic compute cloud (amazon EC2) (2018). https://aws.amazon.com/ec2/. Accessed 2020
8. André, É.: Dynamic clock elimination in parametric timed automata. In: Choppy, C., Sun, J. (eds.) FSFMA, OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, vol. 31, pp. 18–31. Dagstuhl Publishing, Wadern (2013). https://doi.org/10.4230/OASIcs.FSFMA.2013.18
9. André, É., Fribourg, L., Soulat, R.: Merge and conquer: state merging in parametric timed automata. In: Hung, D.V., Ogawa, M. (eds.) ATVA, Lecture Notes in Computer Science, vol. 8172, pp. 381–396. Springer, Berlin (2013). https://doi.org/10.1007/978-3-319-02444-8_27
10. André, É., Lime, D., Roux, O.H.: Integer-complete synthesis for bounded parametric timed automata. In: Bojańczyk, M., Lasota, S., Potapov, I. (eds.) RP, Lecture Notes in Computer Science, vol. 9328, pp. 7–19. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-24537-9_2
11. André, É., Liu, Y., Sun, J., Dong, J.S.: Parameter synthesis for hierarchical concurrent real-time systems. Real-Time Syst. **50**(5–6), 620–679 (2014). https://doi.org/10.1007/s11241-014-9208-6
12. André, É., Soulat, R.: The Inverse Method. FOCUS Series in Computer Engineering and Information Technology. ISTE Ltd. and Wiley, New York (2013)
13. Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: a framework for executing adaptive web-service processes. IEEE Softw. **24**(6), 39–46 (2007). https://doi.org/10.1109/MS.2007.174
14. Ardagna, D., Pernici, B.: Global and local QoS guarantee in web service selection. In: Bussler, C., Haller, A. (eds.) Business Process Management Workshops, vol. 3812, pp. 32–46. IEEE, New York (2005). https://doi.org/10.1007/11678564_4
15. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming **72**(1–2), 3–21 (2008). https://doi.org/10.1016/j.scico.2007.08.001
16. Baresi, L., Guinea, S.: Self-supervising BPEL processes. IEEE Trans. Softw. Eng. **37**(2), 247–263 (2011). https://doi.org/10.1109/TSE.2010.37
17. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. Lectures on Concurrency and Petri Nets, Advances in Petri Nets, Lecture Notes in Computer Science, vol. 3098, pp. 87–124. Springer, Berlin (2003). https://doi.org/10.1007/978-3-540-27755-2_3
18. Bianculli, D., Giannakopoulou, D., Pasareanu, C.S.: Interface decomposition for service compositions. In: ICSE, pp. 501–510 (2011). https://doi.org/10.1145/1985793.1985862
19. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: An approach for QoS-aware service composition based on genetic algorithms. In: Beyer, H., O'Reilly, U. (eds.) GECCO, pp. 1069–1075. ACM, New York (2005). https://doi.org/10.1145/1068009.1068189
20. Cardellini, V., Casalicchio, E., Grassi, V., Presti, F.L., Mirandola, R.: QoS-driven runtime adaptation of service oriented architectures. In: van Vliet, H., Issarny, V. (eds.) FSE, pp. 131–140. ACM, New York (2009). https://doi.org/10.1145/1595696.1595718

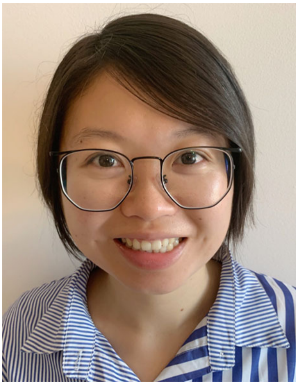
21. Cerný, T., Donahoo, M.J., Pechanec, J.: Disambiguation and comparison of SOA, microservices and self-contained systems. In: RACS, pp. 228–235. ACM (2017). https://doi.org/10.1145/3129676.3129682
22. Ceska, M., Dannenberg, F., Kwiatkowska, M.Z., Paoletti, N.: Precise parameter synthesis for stochastic biochemical systems. In: Mendes, P., Dada, J.O., Smallbone, K. (eds.) CMSB, Lecture Notes in Computer Science, vol. 8859, pp. 86–98. Springer, Berlin (2014). https://doi.org/10.1007/978-3-319-12982-2_7
23. Chen, M., Tan, T.H., Sun, J., Liu, Y., Dong, J.S.: VeriWS: a tool for verification of combined functional and non-functional requirements of web service composition. In: ICSE, pp. 564–567 (2014). https://doi.org/10.1145/2591062.2591070
24. Chen, M., Tan, T.H., Sun, J., Liu, Y., Pang, J., Li, X.: Verification of functional and non-functional requirements of web service composition. In: ICFEM, pp. 313–328 (2013). https://doi.org/10.1007/978-3-642-41202-8_21
25. Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web services description language (WSDL) version 2.0. W3C Recommendation (2007). http://www.w3.org/TR/wsdl20/. Accessed 2020
26. Cimatti, A., Palopoli, L., Ramadian, Y.: Symbolic computation of schedulability regions using parametric timed automata. In: RTSS, pp. 80–89. IEEE Computer Society (2008). https://doi.org/10.1109/RTSS.2008.36
27. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Wang, Z.: Time for statistical model checking of real-time systems. In: CAV, Lecture Notes in Computer Science, vol. 6806, pp. 349–355. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-22110-1_27
28. Daws, C., Yovine, S.: Reducing the number of clock variables of timed automata. In: RTSS, pp. 73–81. IEEE Computer Society (1996). https://doi.org/10.1109/REAL.1996.563702
29. De Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-78800-3_24
30. Eichelmann, T., Fuhrmann, W.F., Trick, U., Ghita, B.V.: Enhanced concept of the TeamCom SCE for automated generated services based on JSLEE. In: Bleimann, U., Dowland, P., Furnell, S., Schneider, O. (eds.) INC, pp. 75–84. University of Plymouth, Plymouth (2010)
31. Ermedahl, A., Sandberg, C., Gustafsson, J., Bygde, S., Lisper, B.: Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In: Rochange, C. (ed.) WCET, OASICS, vol. 6. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Wadern (2007)
32. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: ICSE, pp. 341–350 (2011). https://doi.org/10.1145/1985793.1985840
33. Foster, H.: A Rigorous Approach To Engineering Web Service Compositions. Ph.D. thesis, Imperial College of London (2006)
34. Foster, H., Uchitel, S., Magee, J., Kramer, J.: LTSA-WS: a tool for model-based verification of web service compositions and choreography. In: ICSE, pp. 771–774 (2006). https://doi.org/10.1145/1134408
35. Foundation, A.S.: Apache ODE (2007). http://ode.apache.org/. Accessed 2020
36. Fribourg, L., Lesens, D., Moro, P., Soulat, R.: Robustness analysis for scheduling problems using the inverse method. In: Reynolds, M., Terenziani, P., Moszkowski, B. (eds.) TIME, pp. 73–80. IEEE Computer Society Press, Silver Spring (2012). https://doi.org/10.1109/TIME.2012.10
37. Gao, C., Cai, M., Chen, H.: QoS-aware service composition based on tree-coded genetic algorithm. In: COMPSAC, pp. 361–367. IEEE Computer Society (2007). https://doi.org/10.1109/COMPSAC.2007.174
38. Geebelen, K., Michiels, S., Joosen, W.: Dynamic reconfiguration using template based Web service composition. In: Göschka, K.M., Dustdar, S., Leymann, F., Tosic, V. (eds.) MW4SOC, pp. 49–54. ACM, New York (2008). https://doi.org/10.1145/1462802.1462811
39. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Nielsen, H.F., Karmarkar, A., Lafon, Y.: Simple object access protocol (SOAP) version 1.2. W3C Recommendation (2007). http://www.w3.org/TR/soap12/. Accessed 2020
40. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. Inf. Comput. **251**, 67–90 (2016). https://doi.org/10.1016/j.ic.2016.07.004
41. Hune, T., Romijn, J., Stoelinga, M., Vaandrager, F.W.: Linear parametric model checking of timed automata. J. Log. Algebr. Program. **52–53**, 183–220 (2002). https://doi.org/10.1016/S1567-8326(02)00037-1
42. Jovanović, A., Kwiatkowska, M.Z.: Parameter synthesis for probabilistic timed automata using stochastic game abstractions. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) RP, Lecture Notes in Computer Science, vol. 8762, pp. 176–189. Springer, Berlin (2014). https://doi.org/10.1007/978-3-319-11439-2_14
43. Jovanović, A., Lime, D., Roux, O.H.: Integer parameter synthesis for real-time systems. IEEE Trans. Softw. Eng. **41**(5), 445–461 (2015). https://doi.org/10.1109/TSE.2014.2357445
44. Kraft, S., Pacheco-Sanchez, S., Casale, G., Dawson, S.: Estimating service resource consumption from response time measurements. In: VALUETOOLS, p. 48 (2009). https://doi.org/10.4108/ICST.VALUETOOLS2009.7526
45. Larsen, K.G., Legay, A.: Statistical model checking: past, present, and future. In: Margaria, T., Steffen, B. (eds.) ISoLA, Part I, Lecture Notes in Computer Science, vol. 9952, pp. 3–15 (2016). https://doi.org/10.1007/978-3-319-47166-2_1
46. Le, T.T.H., Palopoli, L., Passerone, R., Ramadian, Y., Cimatti, A.: Parametric analysis of distributed firm real-time systems: a case study. In: ETFA, pp. 1–8. IEEE (2010). https://doi.org/10.1109/ETFA.2010.5641315
47. Lemos, A.L., Daniel, F., Benatallah, B.: Web service composition: a survey of techniques and tools. ACM Comput. Surv. **48**(3), 33:1–33:41 (2016). https://doi.org/10.1145/2831270
48. Li, Y., Tan, T.H., Chechik, M.: Management of time requirements in component-based systems. In: FM, pp. 399–415 (2014). https://doi.org/10.1007/978-3-319-06410-9_28
49. Lloyd, S.P.: Least squares quantization in PCM. IEEE Trans. Inf. Theory **28**(2), 129–136 (1982). https://doi.org/10.1109/TIT.1982.1056489
50. Ma, Y., Zhang, C.: Quick convergence of genetic algorithm for QoS-driven web service selection. Comput. Netw. **52**(5), 1093–1104 (2008). https://doi.org/10.1016/j.comnet.2007.12.003
51. Magee, J., Kramer, J.: Concurrency-State Models and Java programs, 2nd edn. Wiley, New York (2006)
52. Mediouni, B.L., Nouri, A., Bozga, M., Dellabani, M., Legay, A., Bensalem, S.: SBIP 2.0: statistical model checking stochastic real-time systems. In: Lahiri, S.K., Wang, C. (eds.) ATVA, Lecture Notes in Computer Science, vol. 11138, pp. 536–542. Springer, Berlin (2018). https://doi.org/10.1007/978-3-030-01090-4_33
53. Menascé, D.A.: Response-time analysis of composite web services. IEEE Internet Comput. **8**(1), 90–92 (2004). https://doi.org/10.1109/MIC.2004.1260710
54. Merlin, P.M.: A study of the recoverability of computing systems. Ph.D. thesis, University of California, Irvine, CA, USA (1974)
55. Mi, C., Miao, H., Kai, J., Gao, H.: Reliability modeling and verification of BPEL-based Web services composition by probabilistic model checking. In: Song, Y. (ed.) SERA, pp. 149–154. IEEE Com-

puter Society, Silver Spring (2016). https://doi.org/10.1109/SERA.2016.7516140
56. Middleware: the state of microservices survey 2017—eight trends you need to know (2017). https://middlewareblog.redhat.com/2017/12/05/the-state-of-microservices-survey-2017-eight-trends-you-need-to-know/. Accessed 2020
57. Microservices (2018). http://microservices.io/patterns/microservices.html. Accessed 2020
58. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: WWW, pp. 815–824 (2008). https://doi.org/10.1145/1367497.1367607
59. Nguyen, H.G., Petrucci, L., Van de Pol, J.: Layered and collecting NDFS with subsumption for parametric timed automata. In: Lin, A.W., Sun, J. (eds.) ICECCS, pp. 1–9. IEEE Computer Society, Silver Spring (2018). https://doi.org/10.1109/ICECCS2018.2018.00009
60. Ordóñez, A., Alcázar, V., Rendon, O.M.C., Falcarin, P., Corrales, J.C., Granville, L.Z.: Towards automated composition of convergent services: a survey. Comput. Commun. **69**, 1–21 (2015). https://doi.org/10.1016/j.comcom.2015.07.025
61. Pautasso, C.: Restful Web service composition with BPEL for REST. Data Knowl. Eng. **68**(9), 851–866 (2009). https://doi.org/10.1016/j.datak.2009.02.016
62. Quaas, K., Shirmohammadi, M., Worrell, J.: Revisiting reachability in timed automata. In: LICS, pp. 1–12. IEEE Computer Society (2017). https://doi.org/10.1109/LICS.2017.8005098
63. Roussanaly, V., Sankur, O., Markey, N.: Abstraction refinement algorithms for timed automata. In: Dillig, I., Tasiran, S. (eds.) CAV, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 22–40. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-25540-4_2
64. Schmieders, E., Metzger, A.: Preventing performance violations of service compositions using assumption-based run-time verification. In: ServiceWave, pp. 194–205 (2011). https://doi.org/10.1007/978-3-642-24755-2_19
65. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, New York (1986)
66. Simmonds, J., Ben-David, S., Chechik, M.: Guided recovery for web service applications. In: Roman, A., van der Hoek, G.-C. (eds.) SIGSOFT FSE, pp. 247–256. ACM, New York (2010). https://doi.org/10.1145/1882291.1882328
67. Song, Z., Tilevich, E.: Equivalence-enhanced microservice workflow orchestration to efficiently increase reliability. In: Bertino, E., Chang, C.K., Chen, P., Damiani, E., Goul, M., Oyama, K. (eds.) ICWS, pp. 426–433. IEEE, New York (2019). https://doi.org/10.1109/ICWS.2019.00076
68. Sun, J., Liu, Y., Dong, J.S., Liu, Y., Shi, L., André, É.: Modeling and verifying hierarchical real-time systems using stateful timed CSP. ACM Trans. Softw. Eng. Methodol. **22**(1), 3.1–3.29 (2013). https://doi.org/10.1145/2430536.2430537
69. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV, Lecture Notes in Computer Science, vol. 5643, pp. 709–714. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-02658-4_59
70. Swain, S., Niyogi, R.: FESC: functionally equivalent service composition. Internet Things **9**, 100151 (2020). https://doi.org/10.1016/j.iot.2019.100151
71. Tan, T.H., André, É., Chen, M., Sun, J., Liu, Y., Dong, J.S.: Selamat: binary and experiment data (2019). https://sites.google.com/site/automatedsynthesis/home/. Accessed 2020
72. Tan, T.H., André, É., Sun, J., Liu, Y., Dong, J.S., Chen, M.: Dynamic synthesis of local time requirement for service composition. In: Cheng, B.H., Pohl, K. (eds.) ICSE, pp. 542–551. IEEE, New York (2013). https://doi.org/10.1109/ICSE.2013.6606600
73. Tan, T.H., Chen, M., André, É., Sun, J., Liu, Y., Dong, J.S.: Automated runtime recovery for QoS-based service composition. In: WWW, pp. 563–574 (2014). https://doi.org/10.1145/2566486.2568048
74. Tan, T.H., Chen, M., Sun, J., Liu, Y., André, É., Dong, J.S., Xue, Y.: Optimizing selection of competing services with probabilistic hierarchical refinement. In: Visser, W., Williams, L. (eds.) ICSE, pp. 85–95. ACM, New York (2016). https://doi.org/10.1145/2884781.2884861
75. Tari, Z., Bertok, P., Mukherjee, A.: Framework for Modeling, Simulation and Verification of a BPEL Specification, pp. 205–244. Wiley, New York (2013). https://doi.org/10.1002/9781118720103.ch8
76. Traonouez, L.M., Lime, D., Roux, O.H.: Parametric model-checking of stopwatch Petri nets. J. Univ. Comput. Sci. **15**(17), 3273–3304 (2009). https://doi.org/10.3217/jucs-015-17-3273
77. Yu, T., Zhang, Y., Lin, K.J.: Efficient algorithms for web services selection with end-to-end QoS constraints. ACM Trans. Web **1**(1), 6 (2007). https://doi.org/10.1145/1232722.1232728
78. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: Hencsey, G., White, B., Chen, Y.R., Kovács, L., Lawrence, S. (eds.) WWW, pp. 411–421. ACM, New York (2003). https://doi.org/10.1145/775152.775211
79. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. Softw. Eng. **30**(5), 311–327 (2004). https://doi.org/10.1109/TSE.2004.11
80. Zhang, L., Li, B., Chao, T., Chang, H.: On demand web services-based business process composition. In: ICSMC, vol. 4, pp. 4057–4064. IEEE (2003)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Étienne André received his masters degree in computer science from Université Rennes 1 (France) in 2007 and his PhD degree in computer science from École Normale Supérieure de Cachan (France) in 2010. He was a research fellow in 2010–2011 in Prof. Jin-Song Dong's group in the National University of Singapore. From 2011 to 2019, he was an associate professor with Université Paris 13 (France). He has been a full professor with Université de Lorraine (France) since 2019. He is concerned with the formal specification and verification of systems mixing both concurrency and hard real-time constraints. He is particularly interested in parametric timed systems, where timing constants are uncertain or unknown.

Tian Huat Tan is currently a lead data scientist in Data Science and AI Elite team of IBM where he provides consultation to various large enterprises to help them succeed and lead in data science, machine learning and artificial intelligence. Tian Huat is very passionate and dedicated to bringing current state-of-the-art methodologies in data science and artificial intelligence to enterprises, with the purposes of addressing challenging business problems and creating great business value. He has worked in the domain of optimization and machine/deep learning, software engineering and model checking in the past seven years, in both his industry and academic career. He enjoys both the practical and theoretical aspects of these subjects. He has previously obtained his computer science PhD from National University of Singapore, with many of his research results published in highly reputable conferences (e.g., WWW, ICSE, ISSTA, etc.) and journals (e.g., TIFS).



Manman Chen is currently a data scientist in Autodesk where she provides strategies to sales and marketing by leveraging in machine learning and artificial intelligence. She previously obtained her PhD in computer science from National University of Singapore, with many of her research results published in highly reputable conferences (e.g., WWW, ICSE, ISSTA).

Shuang Liu is currently an associate professor at School of Intelligence and Computing, Tianjin University, China. She received bachelors degree in Renmin University of China in 2010 and PhD degree from National University of Singapore in 2015. She worked as a research fellow in SUTD and lecturer in SiT before joining TJU as a faculty member in 2018. Shuang's research interests include software engineering, formal methods and privacy protection.

Jun Sun is currently an associate professor at Singapore Management University (SMU). He received bachelors and PhD degrees in computing science from National University of Singapore (NUS) in 2002 and 2006. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship. He has been a faculty member since 2010 and was a visiting scholar at MIT from 2011 to 2012. Jun's research interests include software engineering, cyber-security and formal methods. He is the co-founder of the PAT model checker. He has published more than 200 articles and conference papers including top conferences in multiple areas.

Yang Liu graduated in 2005 with a Bachelor of Computing (Honours) in the National University of Singapore (NUS). In 2010, he obtained his PhD and started his postdoctoral work in NUS, MIT and SUTD. In 2011, Dr. Liu is awarded the Temasek Research Fellowship at NUS to be the principal investigator in the area of cyber-security. In 2012 fall, he joined Nanyang Technological University (NTU) as a Nanyang Assistant Professor. He is currently a full professor and the Director of the Cybersecurity Lab in NTU. He specializes in software verification, security and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. His work led to the development of a state-of-the-art model checker, Process Analysis Toolkit (PAT). By now, he has more than 300 publications and 6 best paper awards in top-tier conferences and journals. With more than 20 million Singapore dollar funding support, he is leading a large research team working on the state-of-the-art software engineering and cybersecurity problems.

Jin Song Dong completed his PhD degree in computing from University of Queensland in 1995. From 1995 to 1998, he was a research scientist and then senior research scientist at CSIRO in Australia. Since 1998, he has been in the School of Computing at the National University of Singapore (NUS) where he received full professorship in 2016. Currently, he has divided his time between NUS and Griffith University as a professor. His research is in the areas of formal methods, safety and security systems, probabilistic reasoning and trusted machine learning. Jin Song is on the editorial board of ACM Transaction on Software Engineering and Methodology and Formal Aspects of Computing.