Management of Time Requirements in Component-Based Systems

Yi Li[1], Tian Huat Tan[2], and Marsha Chechik[1]

[1] Department of Computer Science, University of Toronto, Canada
[2] ISTD, Singapore University of Technology and Design, Singapore

Abstract. In component-based systems, a number of existing software components are combined in order to achieve business goals. Some of such goals may include system-level (global) timing requirements (GTR). It is essential to refine GTR into a set of component-level (local) timing requirements (LTRs) so that if a set of candidate components collectively meets them, then the corresponding GTR is also satisfied. Existing techniques for computing LTRs produce monolithic representations, that have dependencies over multiple components. Such representations do not allow for effective component selection and repair. In this paper, we propose an approach for building under-approximated LTRs (uLTR) consisting of independent constraints over components. We then show how uLTR can be used to improve the design, monitoring and repair of component-based systems under time requirements. We also report on the implementation of this approach and its evaluation using real-world case studies in Web service composition. The results demonstrate its practical value and advantages over existing techniques.

Keywords: Time requirements, component-based system, service selection, monitoring, error recovery.

1 Introduction

Component-based software design has been widely adopted in practice for its support for separation of concerns, management of complexity and improved reusability. In this paradigm, a number of existing software components are combined to achieve a business goal. Software components usually communicate and interact via a predefined interface specifying the anticipated syntax and behaviors of components. The basic promise of component-based software design is that component services can be used as building blocks for larger integrated systems without the deep knowledge of their internal structures [18]. In other words, system designers can treat interfaces as descriptive abstractions which should be both informative and sufficiently small.

The component-based design methodology has also been successfully applied for time-critical systems such as timed circuits [18], embedded real-time systems [13,22] and Web service compositions. The correctness and reliability of such systems depend not only on the logical computation results but also on their timely response in all circumstances. Hence, it is an important requirement that the end-to-end (*global*) *response time* (GTR) in the composite system is within a particular range (e.g., under 1 second).

Local Time Requirements. The system-level response time clearly depends on the response times of the underlying components. The behavioral correctness of component-based systems is often achieved by establishing contracts between sets of components. If these are met, then the overall behavioral correctness is satisfied. In timed systems, *Local Time Requirements* (LTR) are counterparts of such contracts, establishing constraints on the response times of the individual components. If LTRs are met, then the GTR of the whole system is met as well.

Intuitively, an LTR is a constraint on the parametric response times and the constraint highly depends on the structures of the compositions. Suppose we have a model consisting of two abstract components, C_1 and C_2, taking time t_1 and t_2, respectively. Suppose the GTR for both systems is "produce the response in under k time units". If the two components are sequentially composed, i.e., C_2 takes the output of C_1 as its input, the global response time of the composition is $t_1 + t_2$. In the parallel composition case, both C_1 and C_2 are invoked at the same time, and the output of whoever finishes first is returned. The global response time is thus $min(t_1, t_2)$.

Prior Work. Existing work [21] synthesizes LTRs based on the structure of component compositions and represents them as a *linear real arithmetic* formula in terms of component response times. In the context of our example, LTRs of the two systems would be $t_1 + t_2 \leq k$ and $(t_1 \leq t_2 \Rightarrow t_1 \leq k) \wedge (t_1 > t_2 \Rightarrow t_2 \leq k)$, respectively[1]. An LTR formula typically depends on multiple components. Such a "monolithic" representation has a number of limitations in designing, monitoring and repairing component-based systems. First, at the software design stage, given the LTR of the system, all abstract components appearing in the LTR formula have to be considered together in order to select a suitable combination. This is often infeasible in practice for two reasons: (1) The enumeration of all possible combinations of candidate components is computationally expensive when the number of functionally equivalent components is large [3]. For example, with 5 abstract components and 100 alternatives for each, the total number of possible combinations is 100^5. The situation is even worse when selection has to be done during runtime (online selection). (2) Under the commonly-used *consumer-broker-provider* component service selection model [9] (shown in Fig. 1), the consumer has no direct access to the service, e.g., due to privacy concerns, and service discovery is done using a *discovery agent*. Many Quality of Service (QoS) broker-based service discovery frameworks have been proposed [1,17]. In those frameworks, service search and discovery are delegated to brokers who find suitable services for consumers based on some QoS requirements (e.g., response time, price, availability, etc.) expressed as queries. Such queries can only involve a single component.

Second, violations of time requirements are inevitable during runtime. In complex software systems, the performance of components often varies with time. Sometimes multiple components delay but not all of them are the actual causes of the violation. The monolithic representation of LTRs prevents us from being able to distinguish problematic components and suggest *point-wise* error recovery and adaptation strategies. The only possible recovery strategy is to replace *all* delayed components with function-

[1] Time variables appearing in LTR constraints are implicitly assumed to have real values greater or equal to zero, i.e., $t_1, t_2 \in \mathbb{R}, t_1 \geq 0, t_2 \geq 0$.

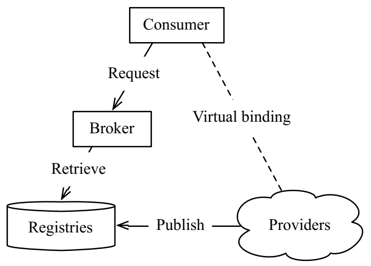


Fig. 1. Illustration of the *consumer-broker-provider* component service selection model [9,1,17]

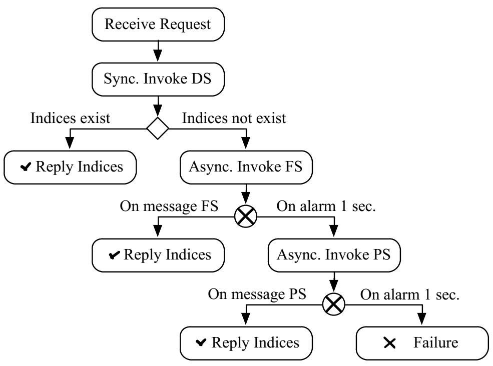


Fig. 2. Workflow diagram of the SMIS example

ally equivalent substitutes that conform to the original timing contracts. Clearly, being able to decompose LTSs and understand the independent timing constraints effectively would yield more options and help find the most efficient recovery strategies.

The ULTR Approach. In this paper, we propose an approach that aims at lifting the above-mentioned limitations of the existing LTR representation by decomposing it into multiple sub-formulas where different abstract components have independent timing contracts. The decomposed constraints under-approximate the original LTR while providing local guarantees on the level of its precision. As a consequence, the component combinations satisfying the under-approximated LTR (denoted by ULTR) also satisfy the original LTR. Recall the parallel composition example. A possible ULTR is $\{\mathcal{B}_1 \equiv t_1 \in [0,\infty) \wedge t_2 \in [0,k], \mathcal{B}_2 \equiv t_1 \in [0,k] \wedge t_2 \in (k,\infty)\}$ which captures the exact same set of software components that meet the timing requirements. The constraints in both sub-formulas $\mathcal{B}_1$ and $\mathcal{B}_2$ treat each component independently, i.e., to check the satisfiability of ULTR, one only needs to look at a single component each time, and once all components satisfy their own contracts, the ULTR is also satisfied.

Given a quantifier-free linear real arithmetic (QF_LRA) formula φ containing only time variables $t_i \in T$ which represent the response times of software components $c_i \in C$, we exploit the power of *Satisfiability Modulo Theories* (SMT) solvers to sample *best under-approximations* of φ, denoted as $BU(\varphi)$. Formula $BU(\varphi)$ is in the Interval (BOX) abstract domain [10], i.e., in the form $\bigwedge_{t_i \in T} l_i \leq t_i \leq u_i$. The key to computing $BU(\varphi)$ is the application of a symbolic optimization procedure which helps find the weakest formula representing a hypercube under the possibly non-convex constraints. The hypercube shaped samples of φ are systematically obtained and aggregated to form a ULTR until it is precise enough. We apply various heuristics according to the structure of φ to ensure fast convergence.

Contributions of This Paper. (1) Given LTRs of a component-based system, we develop a sound method for decomposing these constraints and discharging inter-dependencies over multiple components while providing precision guarantees. (2) We demonstrate the applicability of our method in component selection and its advantages in generation of recovery strategies when compared with the monolithic approach. (3) We evaluate the effectiveness of the uLTR approach in component selection through case studies conducted on real-world Web service compositions.

We implemented our algorithm using the Z3 SMT solver [16] and the symbolic optimization tool OptMathSAT [20] and reported our experience on its applicability in component selection. The candidate Web services were chosen from a publicly available Web service dataset QWS [1]. We also demonstrated that the uLTR model, when adopted in automated error recovery, can help discover repair strategies that are otherwise not possible to find. Supplemental materials including our prototype implementation and LTR constraints for the case studies are available online at `http://www.cs.utoronto.ca/~liyi/ultr/`.

Organization. The rest of the paper is organized as follows. Sec. 2 gives an overview of the uLTR approach using a running example. We present the main algorithm and its applications in Sec. 3 and 4, respectively. Sec. 5 describes the implementation details and empirical evaluation of the effectiveness of our approach. We review related works in Sec. 6 and conclude the paper in Sec. 7.

2 Approach at a Glance

This section illustrates our approach on an example of Web service composition [21].

Stock Market Indices Service (SMIS). SMIS is a paid service to provide updated stock indices to the subscribers. It provides service-level agreement (SLA) to the subscribers stating that it always responds within 3 seconds upon request. The SMIS has three component services: a database service (DS), a free news feed service (FS) and a paid news feed service (PS). The workflow of the composite service is shown in Fig. 2 and is described in the XML-based service composition language BPEL[2].

The SMIS strategy is calling the free service FS before the paid service PS in order to minimize the cost. Upon returning result to the user, SMIS caches the latest results in an external database service provided by DS. Upon receiving the response from DS, if the indices are already available (the `<if>` branch, denoted in Fig. 2 by $\diamond$), they are returned to the user; otherwise, FS is invoked asynchronously. A `<pick>` construct (denoted by $\otimes$) is used here to wait for an incoming response from a previous asynchronous invocation and timeout if necessary. If the response from FS is received within one second, the result is returned to the user. Otherwise, the timeout occurs, and SMIS stops waiting for the result from FS and calls PS instead. Similar to FS, the result from PS is returned to the user if the response from PS is received within one second. Otherwise, it would notify the user regarding the failure of getting stock indices.

LTR Synthesis. Starting with the global timing requirement (GTR) that the composite service must respond within 3 seconds, we use the process of [21] to get the local timing

[2] `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`

$$
\begin{aligned}
\varphi \equiv\ & \neg(0 \le t_{DS} \land 1 \le t_{FS} \land 1 \le t_{PS}) \\
\land\ & (0 \le t_{DS} \land 0 \le t_{FS} \land 0 \le t_{PS}) \\
& \Rightarrow t_{DS} \le 3 \\
\land\ & (0 \le t_{DS} \land 0 \le t_{FS} \\
& \land 0 \le t_{PS} \land t_{FS} \le 1) \\
& \Rightarrow t_{DS} + t_{FS} \le 3 \\
\land\ & (0 \le t_{DS} \land 0 \le t_{PS} \\
& \land 1 \le t_{FS} \land t_{PS} \le 1) \\
& \Rightarrow t_{DS} + t_{PS} \le 2
\end{aligned}
$$

Fig. 3. LTR constraints φ of SMIS

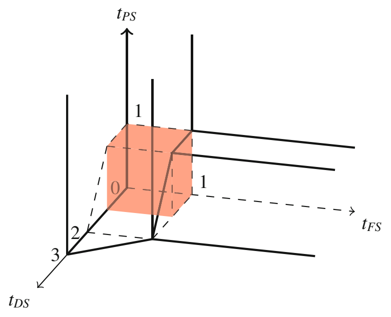


Fig. 4. Feasible region of φ in the 3D space

constraints. The resulting LTR is a quantifier-free linear real arithmetic (QF_LRA) formula φ shown in Fig. 3. It contains three time variables, t_{DS}, t_{FS} and t_{PS}. Geometrically, the feasible region allowed by φ is a non-convex polyhedron in 3-dimensional space as depicted in Fig. 4, and a particular service combination can be represented by a point in the space.

Building uLTR Model via Sampling. To under-approximate φ and guarantee precision at the same time, we greedily sample largest possible hyperrectangles (also referred to as BOXES) from the feasible space constrained by φ by iteratively using an SMT solver. After obtaining each sample, we block the space of that sample from φ so that no portion of φ is explored twice by the solver. For our example, this method proceeds by applying the following operations non-deterministically:

1. Pick a largest possible *hypercube*[3]. Suppose the first sample picked from φ is $([0,1),[0,1),[0,1))$, a shorthand notation for the conjunctive constraint $s_1 \equiv 0 \le t_{DS} < 1 \land 0 \le t_{FS} < 1 \land 0 \le t_{PS} < 1$, which is the largest hypercube at the moment because the three variables cannot be greater than or equal to 1 at the same time under the constraint φ. See the shaded region in Fig. 4.

2. Sample an infinite number of hypercubes at a single step to form a *hyperrectangle* with infinite heights in some dimensions. This allows the algorithm to converge when there are unbounded directions. For example, $([0,1),[1,\infty),[0,1))$ has an infinite height in the t_{FS} direction.

3. Terminate the sampling process when the size of the largest obtained sample is smaller than a predefined precision level $\omega > 0$. More precisely, the algorithm terminates when there is no hypercube of size greater or equal to 2ω left in φ.

As an under-approximation technique, uLTR never returns false positives, i.e., it never erroneously claims that a combination of services satisfies timing requirements. By setting the precision level, our method provides an upper bound for the "local" information loss. For instance, $\omega = 0.1$ ensures that for every misclassified (false negative) point there exists a close enough point (the distance between the projections on some dimension is less than 0.1) which is correctly classified.

[3] A *hyperrectangle* is a generalization of rectangle in an n-dimensional space. A *hypercube* is a special form of a hyperrectangle with an equal height in each dimension.

3 The uLTR Algorithm

3.1 Definitions

Formulas. Let $\mathcal{L}$ be the QF_LRA theory defined as follows:

$$\begin{aligned} \varphi \in \mathcal{L} &::= \mathit{true} \mid \mathit{false} \mid P \wedge P' \mid P \vee P' \\ P, P' \in \mathit{Atoms} &::= c_1 v_1 + \cdots + c_n v_n \bowtie k, \quad n \in \mathbb{N} \\ v_i \in \mathit{Vars} &::= \{v_1, \ldots, v_n\}, \end{aligned}$$

where $c_i, k \in \mathbb{R}$, $\bowtie = \{<, \leq\}$. We use $[\![\varphi]\!]$ to denote the set of all satisfying assignments (models) of φ. A *model* $p : \mathit{Vars} \to \mathbb{R}$ of φ, denoted $p \models \varphi$, is a valuation of the variables of φ such that $\varphi(p) \equiv \mathit{true}$, where $\varphi(p)$ is φ with every occurrence of a variable v replaced by $p(v)$. Geometrically, p is a point in $\mathbb{R}^n$, and in what follows, we use the terms *model* and *point* to refer to p interchangeably. We use $\mathit{Vars}(\varphi)$ to denote the set of all Vars appearing in φ.

Box and Boxes [12]. A set $\mathcal{B} \subseteq \mathbb{R}^n$ is a Box iff it is expressible by a finite system (Cartesian product) of interval constraints. The set of all Box-es of $\mathbb{R}^n$ is denoted by $\mathbb{B}^n$. A set $\mathcal{BS} \subseteq \mathbb{R}^n$ is a Boxes iff there exist Box-es $\mathcal{B}_1, \ldots, \mathcal{B}_k$ such that $\mathcal{BS} = \cup_{i=1}^{k} \mathcal{B}_i$. The set of all sets of Boxes of $\mathbb{R}^n$ is denoted by $\mathbb{BS}^n$.

Let $V = \{v_1, \ldots, v_n\}$ be variables. We assume that each variable is bound to some unique dimension in $\mathbb{R}^n$ and use $\mathsf{form}_V(\mathcal{B})$ to denote the formula $\bigwedge_{v_i \in V} l_i \bowtie v_i \bowtie u_i$ s.t. $p \in \mathcal{B} \Leftrightarrow p \models \mathsf{form}_V(\mathcal{B})$. Similarly, $\mathsf{form}_V(\mathcal{BS})$ denotes the formula $\bigvee_{1 \leq i \leq k} \mathsf{form}_V(\mathcal{B}_i)$. In the rest of the paper, we do not distinguish between the set representation and its corresponding formula representation and abuse the notations $\mathcal{B}$ and $\mathcal{BS}$ to mean $\mathsf{form}_{\mathit{Vars}(\varphi)}$ $(\mathcal{B})$ and $\mathsf{form}_{\mathit{Vars}(\varphi)}(\mathcal{BS})$ respectively.

Precision Level. Assume $\mathcal{BS} \Rightarrow \varphi$. ω is the *precision level* of $\mathcal{BS}$ w.r.t. φ iff

$$\forall p \, \exists p' \, \exists v \cdot p \models \varphi \wedge p \not\models \mathcal{BS} \Rightarrow (p' \models \varphi \Leftrightarrow p' \models \mathcal{BS}) \wedge \mid p(v) - p'(v) \mid \leq \omega.$$

That is, for any false negative p misclassified by $\mathcal{BS}$ there exists another point p' which is correctly captured, and the distance between p and p' in the v direction is less than or equal to ω.

Symbolic Optimization. Let φ be a formula in $\mathcal{L}$. Let f be a *linear objective function*, i.e., a linear term over $\mathit{Vars}(\varphi) = \{v_1, \ldots, v_m\}$, in the form $c_1 v_1 + \cdots + c_m v_m$, where $c_i \in \mathbb{R}$. We say k is the *least upper bound* of f w.r.t. φ and denote it by $\mathsf{Lub}_f(\varphi)$ iff $\varphi \Rightarrow f \leq k$ ($k \in \mathbb{R} \cup \{-\infty, \infty\}$)[4] and there does not exist $k' < k$ where $\varphi \Rightarrow f \leq k'$. The procedure of computing $\mathsf{Lub}_f(\varphi)$ is called *symbolic optimization*.

3.2 Best Under-approximation

We now formalize the notion of *best under-approximation* and describe the algorithm for computing it.

Definitions. Let $\varphi \in \mathcal{L}$. A Box formula $\mathcal{B}$ is *an under-approximation of* φ iff $\mathcal{B} \Rightarrow \varphi$. Let $\mathcal{U}(\varphi)$ be the set of all under-approximations of φ in $\mathbb{B}^n$.

[4] Note that k is ∞ if f is unbounded in φ, and $-\infty$ if φ is unsatisfiable.

```
1: function FBU($\varphi$,$f$)
2:     $\mathcal{B} \leftarrow true$
3:     $\theta \leftarrow \forall\, Vars(\varphi) \cdot (\bigwedge_{v_i \in Vars(\varphi)} \alpha_i \leqslant v_i \leqslant \beta_i) \Rightarrow \varphi$
4:     $\theta' \leftarrow$ QELIM($\theta$)         ▷ Quantifier elimination
5:     $p \leftarrow \mathsf{Lub}_f(\theta')$         ▷ Symbolic optimization computing $p \in [\![\theta]\!]$ that optimizes $f$
6:     for all $v_i \in Vars(\varphi)$ do
7:         $\mathcal{B} \leftarrow \mathcal{B} \wedge (p(\alpha_i) \leq v_i \leq p(\beta_i))$
8:     return $\mathcal{B}$
```

Fig. 5. Algorithm for computing $\mathcal{B} \in BU_f(\varphi)$

```
1: function uLTR ($\varphi, \omega$)
2:     $\mathcal{BS} \leftarrow \varnothing, h \leftarrow \infty, i \leftarrow 0$
3:     $\mathcal{B}_0, h_0 \leftarrow$ MAXCUBE($\varphi$)
4:     while $h_i \geq 2\omega$ do
5:         ASSERT($\neg\mathcal{B}_i$); $i \leftarrow i + 1$
6:         $\mathcal{B}_i, h_i \leftarrow$ MAXCUBE($\varphi$)
7:         if (*) then ▷ non-deterministic
8:             $\mathcal{B}_i \leftarrow$ INFCUBE($\varphi, \mathcal{B}_i$)
9:         $\mathcal{BS} \leftarrow \mathcal{BS} \cup \mathcal{B}_i$
10:    return $\mathcal{BS}$
```

Fig. 6. Iterative hypercube sampling

Let $f : \mathbb{B}^n \to \mathbb{R}$ be a function mapping a BOX formula to a real number. A BOX formula $\mathcal{B}$ is *the best under-approximation of* φ iff $\mathcal{B} \in \mathcal{U}(\varphi)$ and $\forall\, \mathcal{B}' \in \mathcal{U}(\varphi) \cdot f(\mathcal{B}') \leq f(\mathcal{B})$. Let $BU_f(\varphi)$ be the set of all best under-approximations of φ.

Computing Best Under-approximation. Let $\theta \equiv \forall\, Vars(\varphi) \cdot (\bigwedge_{v_i \in Vars(\varphi)} \alpha_i \leqslant v_i \leqslant \beta_i) \Rightarrow \varphi$, where α_i and β_i are real-valued bound variables introduced for each variable v_i in φ. Because the upper and lower bound pairs for all variables uniquely define a BOX formula, we pose the problem of finding $\mathcal{U}(\varphi)$ as computing the set of all satisfying assignments for α_i and β_i in the quantified formula θ. Then we are able to compute the *best* satisfying assignment $BU_f(\varphi)$, which is the optimal BOX formulas in $\mathcal{U}(\varphi)$ w.r.t. f, by calling $\mathsf{Lub}_f(\theta)$. An algorithm FBU is given in Fig. 5. Quantifier elimination has to be applied on θ first to find the quantifier-free equivalent θ' (Line 4) in order to work with symbolic optimization procedures. The function FBU correctly computes the optimal BOX formula $\mathcal{B}$ that makes $\mathcal{B} \Rightarrow \varphi$ valid through finding satisfying assignments of θ, which is supported by Proposition 1.

Proposition 1. *Let $p \in [\![\theta]\!]$. Let ψ be the result of substituting l_i, u_i in $(\mathcal{B} \Rightarrow \varphi)$, for each $v_i \in Vars(\varphi)$, by $p(\alpha_i)$ and $p(\beta_i)$ respectively. Then if θ is satisfiable, ψ is* valid*; otherwise, there does not exist $\mathcal{B} \in \mathbb{B}^n$ such that $\mathcal{B} \Rightarrow \varphi$ is valid.*

3.3 Iterative Hypercube Sampling

We now show how FBU can be used to compute a BOXES formula that under-approximates a given LTR formula φ and ensures a local precision level through iterative hypercube sampling. As in Fig. 6, the uLTR algorithm iteratively samples from φ using an SMT solver and maintains a BOXES formula $\mathcal{BS}$ as the current computed under-approximation of φ. Each new sample $\mathcal{B}_i$ is added to $\mathcal{BS}$ and blocked from the future exploration by asserting $\neg\mathcal{B}_i$ in the SMT solver context (Lines 5 and 9). For example, the grey boxes in Fig. 7 are blocked and the next sample is $\mathcal{B}_5$. The uLTR algorithm makes use of operations MAXCUBE and INFCUBE described below.

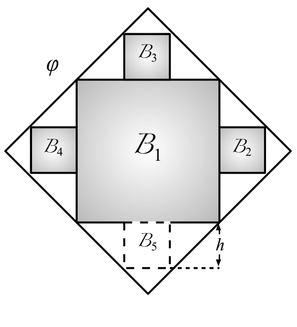


Fig. 7. The iterative sampling process illustration

MAXCUBE(φ). The volume of a hyperrectangle, also known as *hypervolume*, is the product of its heights in all dimensions. Since hypercubes have equal height in all dimensions, i.e., $\beta_i - \alpha_i = h$ for all $v_i \in Vars(\varphi)$ (the value of h is non-negative), we find a hypercube with a maximal volume in φ by asserting an additional constraint $\bigwedge_{v_i \in Vars(\varphi)}(\beta_i - \alpha_i = h)$ and computing a best under-approximation $\mathcal{B}_i \in BU_h(\varphi)$ using the procedure FBU to maximize the height h.

INFCUBE($\varphi, \mathcal{B}_i$). This operation is to ensure convergence when there are dimensions where φ is unbounded. Given a maximal BOX $\mathcal{B}_i$, it tries to relax the constraint in each dimension in a fixed order. Relaxing the constraint $v_i \leq u_i$ is equivalent to sampling an infinite number of hypercubes with the same size as $\mathcal{B}_i$ in the positive direction of dimension v_i. For example, $1 \leq v$ is a relaxation of $1 \leq v \leq 2$ in the positive v direction. If the relaxed BOX $\mathcal{B}'_i$ still under-approximates φ then we can replace $\mathcal{B}_i$ by $\mathcal{B}'_i$.

Correctness and Termination. The algorithm terminates if $h_i < 2\omega$ and the precision level is satisfied. For the example in Fig. 7, if $\omega = \mathsf{height}(\mathcal{B}_5)/2$ then the algorithm terminates after $\mathcal{B}_5$ is sampled since no BOX equal or larger than $\mathcal{B}_5$ left within φ. We now show that when the height of last sample $h < 2\omega$, the precision level of ω is guaranteed. Assume not, then for all mis-classified p there does not exist p' that meets the distance criteria and is correctly classified by $\mathcal{BS}$. Thus, there exists a hypercube $\mathcal{B}'$ centered at p with height 2ω such that $\forall p'' \in \mathcal{B}' \cdot p'' \models \varphi \wedge p'' \not\models \mathcal{BS}$. This contradicts the termination condition $h < 2\omega$, which implies there does not exist such $\mathcal{B}'$.

The correctness of the sampling algorithm trivially follows from the fact that every sample is a BOX formula that implies φ. Therefore, the disjunction of such samples $\mathcal{BS}$ is an under-approximation of φ as well. From the correctness of FBU, the heights h_i of the sampled hypercubes form a non-increasing sequence. The algorithm terminates if INFCUBE is eventually applied.

4 Applications

In this section, we show how uLTR models can be applied in both component selection and runtime error recovery.

Component Selection. Recall that in the *consumer-broker-provider* component service selection model, consumers can only make component-specific search queries through a broker in order to find services they need. Having the property of component independence, uLTR constraints $\mathcal{BS}$ can easily be translated into a sequence of simple service search queries, e.g., "what are the news feed services that have response time less than 0.8 seconds". The broker is able to answer such queries by returning a set of services that satisfy the requirements in the queries [17].

In the SMIS running example, the uLTR computed are the three BOX constraints In the SMIS example, the uLTR computed contains three BOX constraints $\{\mathcal{B}_1 \equiv ([0,1),[0,\infty),[0,1)), \mathcal{B}_2 \equiv ([0,1),[0,1),[1,\infty)), \mathcal{B}_3 \equiv ([1,2),[0,1),[0,\infty))\}$, where intervals in each tuple represent the allowed time ranges for service DS, FS and PS respectively. To reduce the number of remote queries, we could compute the box hull of all BOX constraints first. We first pose three queries, "what are the DS/FS/PS that respond under 2/∞/∞ seconds", which ask for service combinations in the box hull of

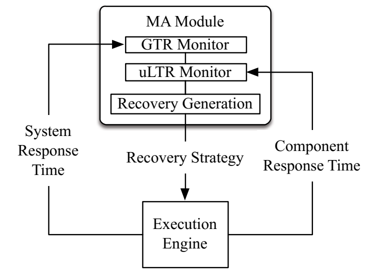


Fig. 8. Component monitoring and recovery framework

```
1: function RECOVERY($\mathcal{BS}, \mathbf{t_e}$)
2:     $distance \leftarrow$ empty map
3:     for all $\mathcal{B} \in \mathcal{BS}$ do
4:         $distance(\mathcal{B}) \leftarrow 0$
5:         for all $[l_i, u_i] \in \mathcal{B}$ do
6:             if $\mathbf{t_e}(i) < l_i \vee \mathbf{t_e}(i) > u_i$ then
7:                 $distance(\mathcal{B}) \leftarrow distance(\mathcal{B}) + 1$
8:     return $\arg\min_{\mathcal{B}' \in \mathcal{BS}}(distance(\mathcal{B}'))$
```

Fig. 9. Algorithm for generating best recovery plans

$\mathcal{B}_1$, $\mathcal{B}_2$ and $\mathcal{B}_3$, i.e., $([0, 2), [0, \infty), [0, \infty))$. In general, we only need *k* remote search queries for compositions of *k* component services. However, the box hull contains infeasible combinations which need to be filtered locally by examining each BOX constraint.

Runtime Adaptation and Recovery. The performance of real-time component-based systems often varies subject to environmental factors over time. It is thus a common practice for such systems to monitor themselves and recover from erroneous behaviors. Fig. 8 depicts a runtime monitoring and recovery framework able to detect violations of timing requirements and suggest efficient recovery strategies. GTR is first used to generate monolithic LTR constraints with which we can compute the initial estimation of the uLTR model. At runtime, the Monitoring and Adaptation (MA) module monitors both the system-level and the component-level response times. The latter are checked against the uLTR model and there are two possibilities when it is violated: (1) The system response time also violates GTR which indicates a real failure. Recovery strategies are then generated and used to instruct the execution engine to recover. (2) GTR is not violated and a false negative is caught. A way to address this problem is to use the false negative to refine the uLTR model and produce a more precise estimation of LTR constraints. The *runtime refinement* is done via a simple MAXCUBE call which computes the largest hypercube containing the false negative point.

uLTR can be used to generate *best* recovery strategies. A *best* recovery strategy is a set of plans requiring a minimum number of component replacements to adapt to the environment changes and put the system back into desired state where the timing requirements are satisfied. Fig. 9 gives an algorithm RECOVERY for generating such strategies. The uLTR model consists of a set of disjoint BOX constraints $\mathcal{BS}$, and the search for recovery plans can be done in a single traverse of this set. Vector $\mathbf{t_e} \in \mathbb{R}^k$ contains the response times for component services during an execution of composite service where the GTR is violated. For each BOX constraint $\mathcal{B}$ in $\mathcal{BS}$, we compute its *distance* (i.e., the number of services that need to be replaced in order to satisfy $\mathcal{B}$) from $\mathbf{t_e}$ by simply comparing the service response time to the corresponding lower and upper bound (Lines 5-7). After the traversal, the function returns a subset of $\mathcal{BS}$ that has the shortest distance from $\mathbf{t_e}$ (Line 8).

In most cases, the recovery plan with the shortest distance is not unique. In the SMIS example, suppose a detected violation has the response time $\mathbf{t_e} = (0.5, 1.5, 1.5)$. There are two BOX constraints in $\mathcal{BS}$ with distance 1 to $\mathbf{t_e}$, i.e., $\mathcal{B}_1 = ([0, 1), [0, \infty), [0, 1))$

and $\mathcal{B}_2 = ([0,1),[0,1),[1,\infty))$, representing two alternative best recovery plans. The execution engine can either replace FS or PS with a substitute that responds under 1 second. With multiple options, the adaptation module can take other QoS parameters (e.g., price) into consideration when making the decision.

5 Implementation and Experiences

5.1 Implementation

We have implemented the uLTR algorithm in C++, using the Z3 SMT solver [16] for satisfiability queries and quantifier elimination. A number of off-the-shelf implementations for computing $\mathsf{Lub}_f(\varphi)$ exist, including SYMBA [14] and OPTMATHSAT [20]. We used the latter. Our implementation accepts an LTR formula written in the standard SMT-LIB2 [6] format and computes the uLTR model as a set of BOXES $\mathcal{BS}$. The source codes of the prototype can be obtained from `http://www.cs.utoronto.ca/~liyi/ultr/`.

In the sampling process, we give priority to hypercubes adjacent to the existing samples so that they can be merged into a larger BOX. We apply INFCUBE periodically and observe the growth of $\mathcal{BS}$. We opportunistically pick the directions where new samples are consecutively obtained, since such directions are often unbounded. These optimizations and heuristics are useful in shortening the time required for convergence.

5.2 Experiences

We performed a series of experiments in order to evaluate the uLTR approach applied for the management of timing requirements during the design and monitoring of component-based systems. Specifically, we aimed to answer the following research questions: **RQ1**: How effective are the uLTR models for the software component selection? **RQ2**: How efficient are the recovery strategies generated by uLTR models?

Subjects. To answer these questions, we designed three case studies on real-world Web service compositions[5] as our subjects which include a stock quotes service (described in Sec. 2), a computer purchasing service and a travel booking service.

Computer Purchasing Service (CPS). The goal of a computer purchasing service (CPS) (e.g., Dell.com) is to allow a user to purchase a computer online using a credit card. CPS uses five component services: Shipment (SS), Logistic (LS), Inventory (IS), Manufacture (MS), and Billing (BS). The global timing requirement of CPS is to respond within 1.6 seconds. LTR computed for CPS contains four time variables.

Travel Booking Services (TBS). The goal of TBS (such as Booking.com) is to provide a combined flight and hotel booking service by integrating two independent existing services. TBS provides an SLA for its subscribed users, promising a response within 1 second after receiving a request. TBS has five component services: user validation (VS), flight (FS), backup flight (FS_{bak}), hotel (HS) and backup hotel (HS_{bak}). LTR for TBS contains four time variables.

[5] Details of the workflows can be found in [21].

Table 1. Statistics of Web services in QWS

Service Types	Quantity	Response Time (ms)			
		MAX.	MIN.	AVG.	STD.
Stock Quotes	13	1,939	67	446	574
Online Data Storage	9	569	144	298	154
Flight Schedule	10	1,212	100	438	330
Hotel Booking	6	440	139	256	104
Online Billing & Payment	13	495	124	105	116
Inventory & Logistic Service	14	4,758	108	545	1,216
Shipping Service	6	278	65	193	84

Dataset. To reflect the actual response times of Web services in our experiments, we used a publicly available Quality of Web Service (QWS) dataset [1]. QWS contains detailed QoS measurements for a set of 2,507 real Web services collected using the Web Service Crawler Engine (WSCE) [2] from public sources including Universal Description, Discovery, and Integration (UDDI) registries[6], search engines, and service portals. Each service has 9 parameters (including response time, availability, throughput, etc.) measured using commercial benchmark tools over a 6-day period in 2008.

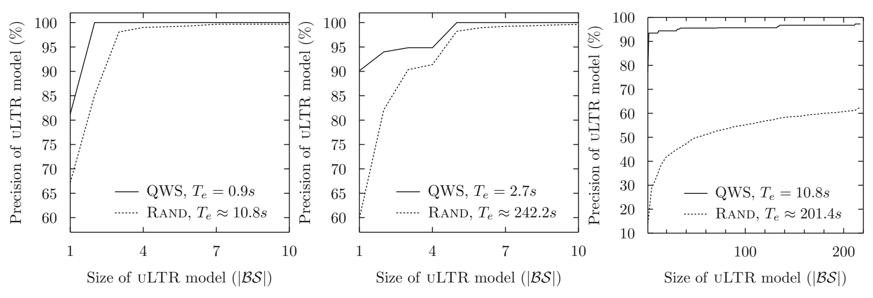
We manually categorized services from the dataset according to their service types[7]. The statistics for each category is given in Table 1. For example, there are 10 flight scheduling services which map to FS and FS_{bak} in the TBS example with an average response time of 438ms and a standard deviation of 330ms. The maximum and minimum response times are 1212ms and 100ms, respectively.

Methodology. For each case study, we compute its uLTR model using the proposed technique setting the precision level ω to 0.05, which should be adjusted accordingly to balance the trade-off between precision and efficiency of the model. We evaluate the quality of the uLTR models in terms of *the percentage of false negatives produced*. Then we simulate a large number of timing requirement violations and examine the recovery strategies generated by the MA module. The experiments were conducted on a computer running Linux with an Intel i5 3.1GHz processor and 4GB of RAM.

RQ1: *Effectiveness of* uLTR *in component selection.* To achieve component-level independence, the uLTR approach loses information on the relationship among components. We evaluate the effectiveness of the uLTR model applied to component selection as its *precision* of approximating the original LTR constraints. Since many of the uLTR models are unbounded (i.e., there is no upper bound for at least one dimension), it is not possible to compute the precision analytically through comparison of the *hypervolume* (the volumes of both uLTR model and LTR model are infinite in this case). Therefore, we study the precision empirically by defining it as the number of service combinations preserved from the original LTR, i.e.,

[6] `http://uddi.org/pubs/uddi-v3.00-published-20020719.htm`

[7] We ignored those services for which the semantics could not be easily inferred from their names or WSDL descriptions.



(a) SMIS (b) TBS (c) CPS

Fig. 10. Precision of uLTR models applied in case studies. T_e is the time taken by enumerating all service combinations. The selection time taken by uLTR is negligible.

$$\text{Precision}(\mathcal{BS}) = \frac{\text{number of combinations satisfied by } \mathcal{BS}}{\text{number of combinations satisfied by } \varphi} \times 100\%.$$

The number of service combinations satisfied by φ is computed by checking the satisfiability of the LTR constraints using Z3. The satisfiability of $\mathcal{BS}$ can be verified by evaluating the Box constraints through simple pairwise comparisons. We used the services in QWS as the target service registry. In order to get statistically significant results, we also randomly generated 10 larger sets (Rand), where each category contains between 16 and 30 services (roughly double the size of QWS), using the Gaussian distribution with the mean and variance recorded in Table 1. The precision results for Rand were obtained by taking the average across the 10 sets.

Results. The experimental results are shown in Fig. 10. The horizontal and vertical axes represent the size of the uLTR model ($|\mathcal{BS}|$) and the precision achieved at that point, respectively. The precision for SMIS (Fig. 10a) and TBS (Fig. 10b) quickly reaches $\sim$100% when the size of $\mathcal{BS}$ increases to 5, without requiring runtime refinement. However, the CPS example exhibits a very different behavior. The uLTR model has good precision results on the QWS set but only achieves $\sim$60% precision on the Rand set (Fig. 10c). A closer look reveals that the structure of the CPS composition imposes much stronger dependencies among component services than the other two. For example, LTR of CPS contains atomic constraints over all four services, and such relationships can hardly be preserved in the Box domain for the dimension-independent nature of Box. A remedy for the information loss during the approximation is *runtime refinement* (Cf. Sec. 4) which is able to restore such information when false negatives are detected during execution.

Furthermore, the time taken by enumerating and evaluating all service combinations (T_e in Fig. 10) increases exponentially as the registry size grows. In contrast, the entire service selection process using the uLTR model was almost instantaneous (<0.01s).

RQ2: *Efficiency of* uLTR *in recovery strategy generation.* The MA module initiates a recovery generation when GTR is violated. Monolithic LTR constraints do not allow pinpointing the actual causes of violations. Without the additional knowledge, the only possible recovery strategy, denoted by LTR, is to replace all delayed components.

Table 2. Comparison of recovery strategies generated by uLTR and LTR models

LTR	uLTR	SMIS (3 comp.)		TBS (4 comp.)		CPS (4 comp.)	
		COUNT	$\bar{D}$(S)	COUNT	$\bar{D}$(S)	COUNT	$\bar{D}$(S)
2	1	5,079	1.29	4,507	0.78	3,224	0.73
	2	72	1.53	644	0.78	1,989	1.08
3	1	4,502	1.63	3,225	1.16	881	0.76
	2	649	2.17	1,926	1.18	3,353	1.18
	3	0	-	0	-	912	1.63
4	1	-	-	232	1.21	139	0.78
	2	-	-	4,919	1.57	1,653	1.20
	3	-	-	0	-	2,962	1.69
	4	-	-	0	-	139	2.22

For each case study, we randomly chose 50 service combinations that originally satisfied the uLTR constraints and simulated service delays by adding a positive random variable D (uniformly distributed between 0.1s and 3s) to some of the response times. We simulated 100 violations for each service combination to get $\sim$15K violations per case study. and compared the length of the best recovery strategies generated using uLTR (denoted by uLTR) with LTR.

Results. The experimental results are summarized in Table 2, in which the columns "LTR", "uLTR" and "COUNT" list the length of the recovery strategies generated using the monolithic LTR approach, the best strategies generated using uLTR models and the number of violations recovered by the corresponding best strategies, respectively. For example, SMIS consists of three components and when two of them are delayed, in 5,079 out of 5,151 cases (98.6%) the system can be recovered by replacing only a single component, whereas LTR would replace both. When all three services are delayed, the best strategies are always shorter in comparison with the naive approach, i.e., no strategy of length 3 is generated. Our experiments clearly show that the best strategies have shorter lengths than the naive approach in the absolute majority of cases.

In Table 2, the column "$\bar{D}$" shows the average delay of component services. The results indicate a correlation between $\bar{D}$ and the length of the best strategies COUNT. That is, the longer the delays, the harder it is to restore the composite system back to the desired state with a small number of replacements. However, the TBS example is a notable exception: it always has best strategies of length at most 2. This has to do with the structure of the composition: if one of the two groups (FS and HS; or FS_{bak} and HS_{bak}) satisfies its requirements, then the overall time requirement is also satisfied. The uLTR approach is able to detect this connection and therefore always produces the shortest repairs.

Recall that we are able to generate multiple best strategies for each violation, but the services required not necessarily exist in the registry. For instance, there is no Inventory Service that responds within 0.1 seconds which is required by some of the best strategies. In our experiment, we have observed that the best strategies could not be executed with the given registry in 31 out of $\sim$15K cases, which is acceptably rare.

In summary, the uLTR models produced for the three case studies are effective in component selection despite the relatively low precision in the CPS example. More importantly, we have also shown that even with the under-approximated models, we are able to generate shorter recovery strategies that were otherwise not possible to find.

Experiences on Scalability. As mentioned, the symbolic optimization tools that we used only accept quantifier-free constraint formulas. The preprocessing step requires quantifier elimination on the linear real arithmetic theory, which is known to be expensive. In practice, the preprocessing of the universally quantified formula θ (Cf. Sec. 3) becomes the bottleneck of the whole sampling process even if it is invoked only once. In our experiment, we gradually increased the number of components in a standard composition structure and observed that the quantifier elimination interface of Z3 is able to handle efficiently compositions with less than 8 components.

Threats to Validity. The first threat is that there are a few random factors in our experiments: the randomly generated Web service registries and GTR violations. To mitigate it, we repeated our experiments a number of times and reported on the averages in the hope to reflect the general cases. The global timing requirements given in the case studies also have an impact on the precision of the uLTR models. Since if the GTR chosen is impractical (either too restrictive or too relaxed), the number of satisfying service combinations can be very skewed (e.g., 0 or everything). In order to mitigate the second threat, we selected GTR so that such cases do not happen in the experiments.

6 Related Work

Computing Under-approximation. Our technique is related to the computation of hyperrectangle-shaped under-approximation for polyhedra. Sankaranarayanan et. al [19] used a random ray shooting technique to find a large enough hyperrectangle over $\mathbb{R}^n$ in convex polyhedra which encodes a conjunction of linear program path conditions. The ray shooting method first finds a random point $\mathbf{t_0}$ within the polyhedron and treats it as a hyperrectangle with zero volume. Then it tries to expand the hyperrectangle while satisfying all constraints by shooting rays to different directions in a fixed order. This process is repeated several times, and the largest hyperrectangle is returned. The under-approximation technique is used to estimate the lower bound for the probability of a set of paths in probabilistic programs. Compared with their method which involves randomness, our algorithm guarantees the maximality of samples and thus ensures the precision level.

Another related problem is computing a maximal inscribed isothetic rectangle in a polygon. An $\Theta(\log n)$ algorithm for computing the maximum area rectangle that has all sides parallel to the coordinate axises and is inscribed in a convex n-gon is given in [4]. This algorithm only works in the 2-dimensional space and has the restriction that the polygon has to be convex. In contrast, by exploiting the power of SMT solvers, our method generalizes to n-dimensional non-convex polyhedra, which is required to express complex timing constraints. Although each single hypercube we computed has equal heights in all dimensions, the disjunctive collection of hypercubes gives us more flexibility in under-approximating non-convex polyhedra.

QoS-Based Service Selection. This work is also related to service selection under QoS constraints. The many techniques proposed for this in the literature can be loosely divided into *service selection with direct access to registries* [7,5,23,3] and *broker-based service discovery* [1,17]. The former assumes the visibility of all concrete services and their QoS attributes (e.g., price, response time and availability) and finds the optimal concrete services. For example, Zeng et. al [23] present an approach that makes use of mixed integer programming (MIP) to dynamically search for the best service combinations under both local and global QoS constraints. [7] formulated service selection as a problem solved by Genetic Algorithms (GA) which allow for non-linear objective functions and provide better scalability.

The broker-based approaches [1,17] delegate the measurement and ranking of QoS parameters to a third-party service discovery agent. This allows users to specify non-functional QoS requirements and find the best services that satisfy the component-level requirements through the broker. [17] introduced a WS-QOS broker architecture which discovers Web services beyond traditional key-word searching. The framework verifies and certifies QoS properties of services and provides services that meet the consumers' requirements through a series of matching, ranking and selection algorithms.

Our work focuses on the timing requirements and is applicable under the *consumer-broker-provider* service selection model which does not assume the availability of all service attributes. We enable point-wise component selection by lifting the dependencies among components. Through sophisticated timing analysis, we extend the broker-based architecture by allowing service discovery to admit not only the component-level but also the system-level global requirements.

Runtime Monitoring and Adaptation. Much work has been done in the area of runtime QoS monitoring and self-adaptation of component-based systems. For example, the KAMI approach [11] combines two basic techniques that support predictions and analysis of QoS properties, namely, *measurement* and *modeling*. KAMI keeps live Bayesian estimator models at runtime for QoS parameter predication and refines the models through the direct measurement of QoS attributes. We adopt a similar approach by modeling the timing requirements using an under-approximation and making the model progressively more accurate when discrepancies are detected at runtime. The difference of our work is that we use the uLTR model to generate *best* adaptation strategies while the approaches in [11] use a predefined violation handler.

A number of other service monitoring and adaptation frameworks including MOSES [8] and VieDAME [15] use specific service selection algorithms to choose the optimal replacement when a service failure is found. None of them address the problem of "*best* adaptation strategy" in terms of the number of services to replace when there are multiple delays of component services. Their techniques in replacement optimization is orthogonal to our approach and can be used to choose the optimal one when multiple *best* strategies are generated.

7 Conclusions and Future Work

In this paper, we presented the uLTR approach which decomposes the monolithic representation of LTR constraints into independent timing contracts over software components. Our method is based on an iterative sampling algorithm using SMT solvers. The uLTR algorithm computes the under-approximation of LTR in the Box domain which guarantees local precision level. We showed how the uLTR models can be applied in component selection and runtime adaption strategy generation under timing requirements. Our experience demonstrates the applicability and effectiveness of the uLTR approach in real-world service compositions.

We see many avenues for future work. First, we would like to extend our approach to allow handling requirements containing QoS attributes other than time. This requires defining an automatic synthesis procedure for those requirements and their efficient encoding in linear real arithmetic. Another direction is generalizing the *best under-approximation* algorithm to allow sampling of arbitrary hyperrectangles. This relies on the development of a non-linear symbolic optimization procedure. Finally, we are interested in lifting the scalability limitations by avoiding quantifier elimination through approximating best under-approximation computations.

Acknowledgement. We are grateful for the valuable discussions and helpful comments from Aws Albarghouthi and Zachary Kincaid.

References

1. Al-Masri, E., Mahmoud, Q.H.: QoS-based Discovery and Ranking of Web Services. In: Proc. of ICCCN 2007, pp. 529–534. IEEE (2007)
2. Al-Masri, E., Mahmoud, Q.H.: Investigating Web Services on the World Wide Web. In: Proc. of WWW 2008, pp. 795–804 (2008)
3. Alrifai, M., Skoutas, D., Risse, T.: Selecting Skyline Services for QoS-Based Web Service Composition. In: Proc. of WWW 2010, pp. 11–20. ACM (2010)
4. Alt, H., Hsu, D., Snoeyink, J.: Computing the Largest Inscribed Isothetic Rectangle. In: Proc. of CCCG 1995, pp. 67–72 (1995)
5. Ardagna, D., Pernici, B.: Global and Local QoS Guarantee in Web Service Selection. In: Bussler, C.J., Haller, A. (eds.) BPM 2005. LNCS, vol. 3812, pp. 32–46. Springer, Heidelberg (2006)
6. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, The University of Iowa (2010), `http://www.SMT-LIB.org`
7. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: An Approach for QoS-aware Service Composition Based on Genetic Algorithms. In: Proc. GECCO 2005, pp. 1069–1075 (2005)
8. Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Lo Presti, F., Mirandola, R.: Moses: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems. IEEE TSE (2012)
9. Carminati, B., Ferrari, E., Hung, P.C.: Exploring Privacy Issues in Web Services Discovery Agencies. IEEE Security & Privacy 3(5), 14–21 (2005)
10. Cousot, P., Cousot, R.: Static Determination of Dynamic Properties of Programs. In: Proc. of the Colloque sur la Programmation (1976)
11. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model Evolution by Run-time Parameter Adaptation. In: Proc. of ICSE 2009, pp. 111–121. IEEE (2009)
12. Gurfinkel, A., Chaki, S.: Boxes: A Symbolic Abstract Domain of Boxes. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 287–303. Springer, Heidelberg (2010)
13. Isovic, D., Norström, C.: Components in Real-time Systems. In: Proc. of ICRTCSA 2002 (2002)

14. Li, Y., Albarghouthi, A., Gurfinkel, A., Kincaid, Z., Chechik, M.: Symbolic Optimization with SMT Solvers. In: Proc. of POPL 2014 (2014)
15. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive Monitoring and Service Adaptation for WS-BPEL. In: Proc. of WWW 2008, pp. 815–824. ACM (2008)
16. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. Rajendran, T., Balasubramanie, P., Cherian, R.: An Efficient WS-QoS Broker Based Architecture for Web Services Selection. Int. J. of Computer Applications 1(9), 110–115 (2010)
18. Salah, R.B., Bozga, M., Maler, O.: On Timed Components and Their Abstraction. In: Proc. of SAVCBS 2007, pp. 63–71 (2007)
19. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. In: Proc. of POPL 2013, New York, NY, USA, pp. 447–458 (2013)
20. Sebastiani, R., Tomasi, S.: Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ Cost Functions. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 484–498. Springer, Heidelberg (2012)
21. Tan, T.H., André, E., Sun, J., Liu, Y., Dong, J.S., Chen, M.: Dynamic Synthesis of Local Time Requirement for Service Composition. In: Proc. of ICSE 2013, pp. 542–551 (2013)
22. Wang, S., Rho, S., Mai, Z., Bettati, R., Zhao, W.: Real-time Component-based Systems. In: Proc. of RTETAS 2005, pp. 428–437 (2005)
23. Zeng, L., Benatallah, B., Ngu, A.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware Middleware for Web Services Composition. IEEE TSE 30(5), 311–327 (2004)