SMT-based Verification of LTL Specifications with Integer Constraints and its Application to Runtime Checking of Service Substitutability

Marcello M. Bersani
Politecnico di Milano
Milano, Italy
bersani@elet.polimi.it

Luca Cavallaro
Politecnico di Milano
Milano, Italy
cavallaro@elet.polimi.it

Achille Frigeri
Politecnico di Milano
Milano, Italy
frigeri@elet.polimi.it

Matteo Pradella
CNR IEIIT-MI
Milano, Italy
pradella@elet.polimi.it

Matteo Rossi
Politecnico di Milano
Milano, Italy
rossi@elet.polimi.it

***Abstract*—An important problem that arises during the execution of service-based applications concerns the ability to determine whether a running service can be substituted with one with a different interface, for example if the former is no longer available. Standard Bounded Model Checking techniques can be used to perform this check, but they must be able to provide answers very quickly, to avoid that the check may affect the operativeness of the application, instead of aiding it. The problem becomes even more complex when *conversational services* are considered, i.e., services that expose operations that have Input/Output data dependencies among them. In this paper we introduce a formal verification technique for an extension of Linear Temporal Logic that allows users to include in formulae constraints on integer variables. This technique applied to the substitutability problem for conversational services is shown to be considerably faster and with smaller memory footprint than existing ones.**

***Keywords*-Bounded Model Checking, SMT solver, Service-Oriented Architectures.**

I. Introduction

Service Oriented Architectures (SOAs) are a flexible set of design principles that promote interoperability among loosely coupled services that can be used across multiple business domains. In this context applications are typically composed of services made available by third-party vendors. Thus, an organization does not have total control of every part of the application, hence failures and service unavailability should be taken into account at runtime. On the other hand, during the application execution new services might become available, that enable new features or provide equivalent functionalities with better quality. Therefore the ability to support the evolution of service compositions, for example by allowing applications to substitute existing services with others discovered at runtime, becomes crucial.

Most of the frameworks proposed in recent years for the runtime management of service compositions make the assumption that all semantically equivalent services agree on their interface [1], [2]. In the practice this assumption turns out to be unfounded. The picture is further complicated when one considers *conversational services*, i.e., services that expose operations with input/output data dependencies among them. In fact, in this case the composition must deal with *sequences* of operation invocations, i.e., the *behavior protocol*, instead of single, independent, ones.

The *substitutability* problem is the problem of deciding when a service can be dynamically substituted by another one discovered at runtime. In [3], [4] authors propose an approach based on Bounded Model Checking (BMC) techniques. Even if the approach proves to be quite effective, the Propositional Satisfiability (SAT) problem, on which the standard encoding of BMC relies, requires to deal with lengthy constraints, which typically limits the efficiency of the analysis phase. In the setting of the runtime management of service compositions this is not acceptable, as delays incurred when deciding whether services are substitutable or not can hamper the operativeness of the application.

In this paper, we introduce a verification technique, based on Satisfiability Modulo Theories (SMT), for an extension of Propositional Linear Temporal Logic with Both past and future operators (PLTLB). This extension, called CLTLB(DL), allows users to define formulae including Difference Logic (DL) constraints on time-varying integer variables.

Our SMT-based verification technique has two main advantages:

- unlike in the standard approach to BMC problems, arithmetic domains are not approximated by means of a finite representation, which proves to be particularly useful in the service substitutability problem;
- the implemented prototype is shown to be considerably faster and with smaller memory footprint than existing ones based on the propositional encoding of BMC, due to the conciseness of our solution.

The technique exploits decidable arithmetic theories supported by many SMT solvers [5] to natively deal with integer variables (hence, with an infinite domain). This allows us to decide larger substitutability problems than before, in significantly less time: the response times of our prototype tool make it usable also in a runtime checking setting.

This paper is structured as follows: Section II introduces the issues underlying the runtime checking of service substitutability; Sections III and IV present, respectively, CLTLB(DL) and its SMT-based encoding for verification purposes; Section V explains how the approach works on

a case study, and Section VI discusses some experimental results. Finally, Section VII presents some related works.

II. Substitutability Checking of Conversational Services

The approach presented in [3] enables service substitution through the automatic definition of suitable *mapping scripts*. These map the sequences of operations that the client is assuming to invoke on the *expected service* into the corresponding sequences made available by the *actual service* (i.e., the service that will be actually used). Mapping scripts are automatically derived given:

- a description of service interfaces in which input and output parameters are associated with each service operation,
- the behavioral protocol associated with each service, described through an automaton.

The mapping between an expected and an actual service assumes that the *compatibility between data* has been previously defined. This relation allows us to map data of different services to the same label. For the sake of simplicity, here we assume that data are compatible if they are called the same way (more sophisticated compatibility relationships are explored in [6]).

Given this compatibility definition, we say that a sequence of operations seq_{exp} in the automaton of the expected service is *substitutable* by another sequence of operations seq_{act} in the automaton of the actual service if a client designed to use the expected service sequence can use the actual service sequence without noticing the difference. This happens when the actual operations require as input at most all of the data provided as input to the expected operations and return at least all the data the expected sequence provides as output.

The rationale of this definition can be understood by considering the service substitution process. In this process a client is designed to interact with an expected service and, therefore, it assumes to invoke the expected service operations, providing for each invocation the data required by the operation and awaiting as output the data provided by the operation. When the expected service is substituted by an actual service, in order for the client to be unaware of the change, the actual service should be able to work with the data provided by the client as if to the expected service and should return the data the client is awaiting from the expected service.

The formal model of substitutability allows us to build a reasoning mechanism based on temporal logic that, given an expected service sequence, returns a corresponding actual service sequence. It includes the behavioral protocols of both the expected and the actual services represented as Labeled Transition Systems (LTS) and formalized in temporal logic, in which each transition is labeled with the associated operation. Input and output parameters of each operation

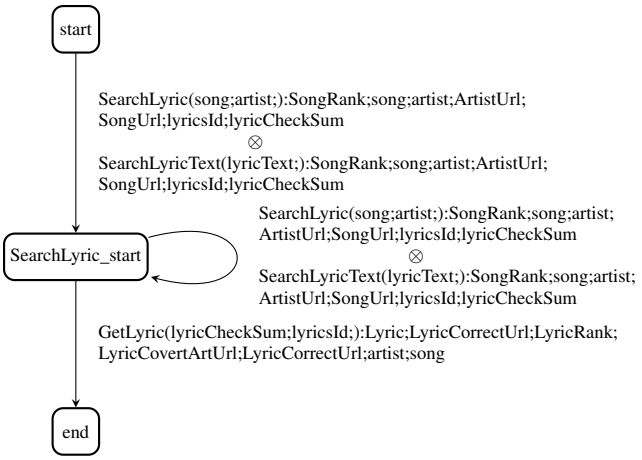


Figure 1. LTS of the ChartLyrics service of Section V ($\otimes$ denotes that the operations are on different transitions).

are also part of the model (Figures 1 and 2 show the LTSs of two services discussed in Section V).

In order to model the substitution process and to keep track of the data exchanged we maintain two kinds of counters:

- seen, which is used to check that the actual service can work using a subset of the input data provided by the client to the expected service,
- needed, which is used to check that the actual service can provide a superset of the data the client expects to receive as output of the expected service.

The model includes an instance of seen (resp. needed) for each type of data that can be used as input (resp. output) parameter for an operation. Each time an operation of the expected service is invoked, the instances of seen for its input parameters and those of needed for its output parameter are all incremented by one.

To illustrate this mechanism, consider the services depicted in Figures 1 and 2, which represent, respectively, the expected and the actual service (the two services are presented in more detail in Section V; in this Section we focus only on the aspects that are relevant to the example at hand). Operation *SearchLyric* of the expected service of Fig. 1 has two input parameters, *song* and *artist*, and five output parameters, *SongRank*, *song*, *artist*, *ArtistUrl*, *SongUrl*, *lyricsId* and *lyricCheckSum*. After its invocation, *seen(song)* and *seen(artist)* are incremented by 1. The same increment takes place for *needed(SongRank)*, *needed(song)*, *needed(artist)*, *needed(ArtistUrl)*, *needed(SongUrl)*, *needed(lyricsId)* and *needed(lyricCheckSum)*. Conversely, when an operation of the actual service is invoked, the instances of the seen counter for each input parameter and those of the needed counter for each output parameter are all decremented by one. For example, when operation *checkSongExists* of the actual service of Fig. 2 is invoked, *seen(song)*,

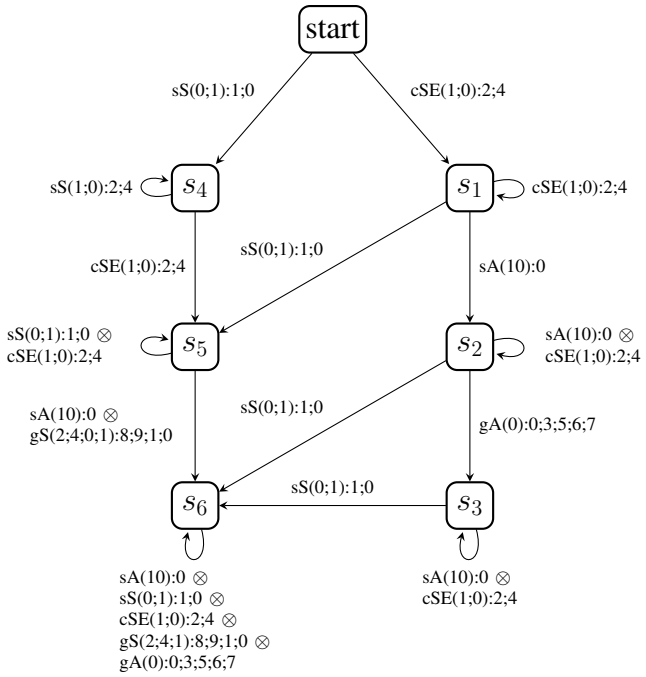


Figure 2. LTS of the *LyricWiki* service discussed in Section V. **Operations**: searchSongs (sS), checkSongExists (cSE), searchArtists (sA), getArtist (gA), getSong (gS). **Parameters**: artist (0), song (1), lyricsId (2), item (3), lyricCheckSum (4), SongUrl (5), year (6), album (7), LyricCorrectUrl (8), Lyrics (9), lyricText (10) (⊗ denotes that the operations are on different transitions).

seen(artist), *needed(lyricsId)* and *needed(lyricCheckSum)* are decremented by 1 (and consequently run to 0).

To conform to the notion of substitutability of expected and actual sequences of operations presented above, an actual service operation can be invoked only if the seen counter for each of its input parameters is ≥ 0 (i.e. the input parameters have been provided by a client assuming to invoke some operations on the expected service). When the value of a needed counter is 0 it means that the actual service provided enough instances of a certain type of data to fulfill client requests. If, on the other hand, the actual service provides more instances of a type of data than those requested, then the corresponding needed counter is < 0.

In case the expected service operation sequence analyzed is substitutable by one in the actual service, a mapping script is generated and then interpreted by an *adapter* that intercepts all service requests issued by the client and transforms them into some requests the actual service can understand. Fig. 3 shows the placement of adapters into the infrastructure architecture and highlights their nature of intermediaries (see [3] for details).

III. A LOGIC FOR TIME-VARYING COUNTERS

In order to deal with time-varying counters over actual domains (such as seen and needed discussed in Section II), we introduce an extension of Linear-time Temporal Logic

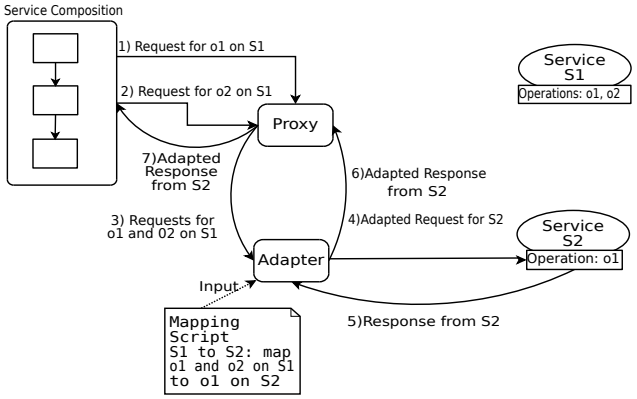


Figure 3. The adaptation runtime infrastructure.

with past operators and non-quantified first order integer variables. The language we consider, denoted CLTLB(DL), is an extension of PLTLB which combines pure Boolean atoms and formulae with terms defined by DL constraints. Counters can naturally be represented by integer variables over the whole domain without any approximation due to a propositional encoding. In [7] we prove the decidability of the satisfiability problem in more general cases.

Difference Logic is the structure $\langle \mathbb{Z}, =, (<_d)_{d\in\mathbb{Z}} \rangle$, where each $<_d$ is a binary relation defined as

$$x <_d y \Leftrightarrow x < y + d.$$

The notations $x < y$, $x \leq y$, $x \geq y$, $x > y$ and $x = y + d$ are abbreviations for $x <_0 y$, $x <_0 y \vee x = y$, $\neg(x <_0 y)$, $\neg(x <_0 y \vee x = y)$ and $y <_{d-1} x \wedge x <_{d+1} y$, respectively.

Let AP the set of Atomic Propositions and V the set of variables and $x \in V$. By defining *arithmetic temporal terms* (a.t.t.'s) as

$$\alpha := x \mid \mathbf{X}\alpha \mid \mathbf{Y}\alpha,$$

the syntax of (well formed) formulae of CLTLB(DL) is:

$$\phi := p \mid \alpha \sim \alpha \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{X}\phi \mid \mathbf{Y}\phi \mid \mathbf{Z}\phi \mid \phi\mathbf{U}\phi \mid \phi\mathbf{S}\phi,$$

where $p \in AP$, α's are a.t.t.'s, $\sim \in \{=, (<_d)_{d\in\mathbb{Z}}\}$, $\mathbf{X}$ is the "next" operator, $\mathbf{Y}$ and $\mathbf{Z}$ are "previous" operators, $\mathbf{U}$ and $\mathbf{S}$ are "until" and "since" operators. The *depth* $|\alpha|$ of an a.t.t. is the total amount of temporal shift needed in evaluating α:

$$|\mathbf{X}(\alpha)| = |\alpha| + 1,$$
$$|\mathbf{Y}(\alpha)| = |\alpha| - 1,$$

where $|x| = 0$. Depth extends naturally to formulae as the minimum depth of its a.t.t.'s.

The semantics of a formula ϕ of CLTLB(DL) is defined w.r.t. a linear time structure $\pi_\sigma = (S, s_0, I, \pi, \sigma, L)$, where S is the set of states, s_0 is the initial state, $I : [|\phi|, -1] \times V \to \mathbb{Z}$ is an assignment of variables, $\pi \in s_0 S^\omega$ is an *infinite path*, $\sigma : \mathbb{N} \times V \to \mathbb{Z}$ is a sequence of valuations, and $L : S \to 2^{AP}$ is the labeling function. Function I allows a valuation

of variables to be defined also for instants preceding zero and then to be extended to a.t.t.'s. Indeed, if α is such a term, x is the variable in α, s_i is a state along the sequence, and σ^i is a shorthand for $\sigma(i,\cdot)$, then:

$$\sigma^i(\alpha) = \begin{cases} \sigma^{i+|\alpha|}(x), & \text{if } i+|\alpha| \geq 0; \\ I(i+|\alpha|, x), & \text{if } i+|\alpha| < 0. \end{cases}$$

Given a linear time structure π_σ, the semantics of a formula ϕ is recursively defined as:

$$\pi^i_\sigma \models p \Leftrightarrow p \in L(s_i) \text{ for } p \in AP$$
$$\pi^i_\sigma \models (\alpha_1 \sim \alpha_2) \Leftrightarrow \sigma^{i+|\alpha_1|}(x_{\alpha_1}) \sim \sigma^{i+|\alpha_2|}(x_{\alpha_2})$$
$$\pi^i_\sigma \models \neg\phi \Leftrightarrow \pi^i_\sigma \not\models \phi$$
$$\pi^i_\sigma \models \phi \wedge \psi \Leftrightarrow \pi^i_\sigma \models \phi \text{ and } \pi^i_\sigma \models \psi$$
$$\pi^i_\sigma \models \mathbf{X}\phi \Leftrightarrow \pi^{i+1}_\sigma \models \phi$$
$$\pi^i_\sigma \models \mathbf{Y}\phi \Leftrightarrow \pi^{i-1}_\sigma \models \phi \wedge i > 0$$
$$\pi^i_\sigma \models \mathbf{Z}\phi \Leftrightarrow \pi^{i-1}_\sigma \models \phi \vee i = 0$$
$$\pi^i_\sigma \models \phi\mathbf{U}\psi \Leftrightarrow \begin{cases} \exists j \geq i : \pi^j_\sigma \models \psi \wedge \\ \pi^n_\sigma \models \phi \forall\, i \leq n < j \end{cases}$$
$$\pi^i_\sigma \models \phi\mathbf{S}\psi \Leftrightarrow \begin{cases} \exists 0 \leq j \leq i : \pi^j_\sigma \models \psi \wedge \\ \pi^n_\sigma \models \phi \forall\, j < n \leq i \end{cases}$$

where x_{α_i} is the variable that appears in α_i and $\sim$ is any relation in DL. The $\mathbf{R}$ and $\mathbf{T}$ operators and their semantics over infinite paths, can be defined as usual: $\phi\mathbf{R}\psi \equiv \neg(\neg\phi\mathbf{U}\neg\psi)$ and $\phi\mathbf{T}\psi \equiv \neg(\neg\phi\mathbf{S}\neg\psi)$. A formula $\phi \in$ CLTLB(DL) is *satisfiable* if there exists a linear time structure $(S, s_0, I, \pi, \sigma, L)$ such that $\pi^0_\sigma \models \phi$ (in which case π_σ is a model of ϕ).

Unfortunately, CLTLB(DL) is too expressive in the sense that the satisfiability problem can be proven to be highly undecidable [8]. However, the satisfiability and the model checking problems for a CLTLB(DL) formula ϕ interpreted over *k-partial* valuations σ_k (i.e., computations in which the value of the counters is considered only up to k plus the *maximum* depth of the subformulae of ϕ) is shown to be decidable [7] when suitable restrictions are considered. The two problems reduce to the satisfiability and the model checking problems, respectively, over bounded paths, of length equal to k. As in the standard BMC (of a property ϕ) the goal is to look for a finite initialized path of the system that is a witness of wrong behaviors, i.e., a path along which the negation of the property ϕ holds. When the finite path of length k admits a loop, it represents an infinite periodic behavior; conversely, when a loop does not exist, the finite path represents all its possible infinite extensions. Formally, finite paths π are words of states s_i which are possibly periodic; if the loop exists, we have $\pi = us_lvs_l$, with $u = s_0 \cdots s_{l-1}$, $v = s_{l+1} \cdots s_k$ and $l \leq k$, and π is a finite representation of infinite path $u(s_lv)^\omega$. In addition, the *values* of the variables up to state s_k are depicted by a bounded representation $\widehat{\sigma}_k$ of σ_k. Arithmetic DL constraints may be part of the (possibly periodic) finite structure $\pi_{\widehat{\sigma}_k}$ and, thus, are represented through a finite prefix of length k. According to [7], [9], we are allowed to use a proper bounded semantics to state reachability properties on that part of the system involving a counting mechanism (i.e., $\mathbf{X}x = y + 1$, where x, y are variables). Note that over finite acyclic paths, the equivalence $\phi\mathbf{R}\psi \equiv \neg(\neg\phi\mathbf{U}\neg\psi)$ and $\phi\mathbf{T}\psi \equiv \neg(\neg\phi\mathbf{S}\neg\psi)$ no longer holds. Then, $\mathbf{R}$ (and symmetrically $\mathbf{T}$) is redefined, for finite acyclic paths, as in [10]:

$$\pi^i_{\widehat{\sigma}_k} \models_k \phi\mathbf{R}\psi \Leftrightarrow$$
$$\exists\, i \leq j \leq k, \pi^j_{\widehat{\sigma}_k} \models_k \phi \wedge \pi^n_{\widehat{\sigma}_k} \models_k \psi \forall\, i \leq n \leq j.$$

Then, the (existential) reachability problem over infinite paths π_{σ_k} endowed with a k-partial valuation σ_k, $\pi_{\sigma_k} \models \phi$, can be reduced to the bounded (existential) reachability problem over finite (possibly cyclic) paths $\pi_{\widehat{\sigma}_k}$, $\pi_{\widehat{\sigma}_k} \models_k \phi$:

Theorem 1 ([7]). *Let ϕ be a* CLTLB(DL) *formula. If there exists $k > 0$, a finite path π and a finite assignment $\widehat{\sigma}_k$ such that $\pi_{\widehat{\sigma}_k} \models_k \phi$ then the formula ϕ is satisfiable over k-partial valuations.*

Then, we are allowed to correctly verify the satisfiability of CLTLB(DL) formulae over k-partial valuations and also to realize a bounded model checking of systems involving DL constraints. Particularly, when a counting mechanism is defined, reachability properties of values of variables along paths of finite length can be verified. Obviously, if the reachability property does not hold within k, then k can be refined and augmented. As explained later in Section VI, the substitutability problem can be solved by means of a BMC approach by correctly estimating an upper bound of k. This is done by using a suitable heuristic based on the sizes of the automata describing services and on the length of the traces of invocations. For these reasons, the substitutability problem, which requires to analyze a counting mechanism over finite paths of invocations of service operations, can be easily encoded into a bounded reachability problem.

IV. Encoding of Bounded Reachability Problem

In this section the bounded reachability problem is encoded as satisfiability of a Quantifier Free Integer Difference Logic formula with Uninterpreted Function and predicate symbols (QF-UFIDL). Such a logic is shown to be decidable, and its satisfiability problem to be NP-complete (for example by applying the Nelson-Oppen Theorem). The QF-UFIDL encoding results to be more succinct and expressive than the Boolean one: lengthy propositional constraints are substituted by more concise DL constraints and arithmetic (infinite) domains do not require an explicit finite representation. These facts, considering also that the satisfiability problem for QF-UFIDL has the same complexity as SAT, make the SMT-based approach particularly efficient to solve the runtime substitutability problem, as demonstrated

by the performance results shown in Section VI. In the seminal work of Biere et al. [9], the BMC is reduced to a pure propositional satisfiability problem. This approach, and further refinements [10], [11], [12], has been already implemented in the bounded satisfiability checker tool Zot (http://home.dei.polimi.it/pradella/).

A. Encoding the Time

As discussed before, the BMC problem amounts to look for a finite representation of infinite (possibly periodic) paths. The SAT-based approach encodes finite paths [9] by means of $2k+3$ propositional variables. The time instant at which the periodic suffix starts is defined by the *loop selector variables* $l_0, l_1, \ldots l_k$: l_i holds if and only if the loop starts at instant i, i.e., s_i is the successor of s_k. Then, the truth (of atomic propositions) in s_i and s_k, defined by the labeling function L defined in Section III, must be the same. Further propositional variables, $inLoop_i$ $(0 \leq i \leq k)$ and $loopEx$, respectively, mean that time instant i is inside a loop and that there actually exists a loop.

The same temporal behavior can be defined by means of *one* QF-UFIDL formula involving only *one* integer *loop-selecting* variable $\boldsymbol{loop} \in \mathbb{Z}$:

$$\bigwedge_{i=1}^{k} \big((\boldsymbol{loop} = i) \Rightarrow (L(s_{i-1}) = L(s_k))\big).$$

The QF-UFIDL encoding is more concise: it does not require $2k+3$ Boolean variables (l_i, $inLoop_i$ and $loopExists$). A value of $\boldsymbol{loop}$ between 1 and k defines if there exists a loop and its position; it does not depend on the k parameter.

B. Encoding the Arithmetic Temporal Terms

Since CLTLB(DL) formulae consist also of a.t.t.'s, we need to define a suitable encoding for them. An *arithmetic formula function*, i.e. an uninterpreted function $\boldsymbol{\tau} : \mathbb{Z} \to \mathbb{Z}$, is associated with each arithmetic temporal subterm of Φ. Let τ be such a subterm, then the arithmetic formula function associated with it (denoted by the same name but in written in bold face), is recursively defined w.r.t. the sequence of valuations σ as:

τ	$0 \leq i \leq k$
x	$\boldsymbol{x}(i) = \sigma^i(x)$
$\mathbf{X}\alpha$	$\boldsymbol{\tau}(i) = \boldsymbol{\alpha}(i+1)$
$\mathbf{Y}\alpha$	$\boldsymbol{\tau}(i) = \boldsymbol{\alpha}(i-1)$

This semantics is well-defined between 0 and k thanks to the initialization function I.

C. Encoding the Propositional Terms

The propositional encoding is inspired from the one studied in [10], but it is deeply revised to take also into account relations over a.t.t.'s. In the case of the Boolean encoding, the semantics of a PLTLB formula Φ is defined w.r.t. the truth value of its subformulae by means of Boolean variables t associated with each of them, for all $0 \leq i \leq k+1$: if t_i holds then subformula t holds at instant i. Instant $k+1$ is introduced to more easily represent the instant in the past when the periodic behavior starts. The propositional encoding of a CLTLB(DL) formula Φ is defined as for PLTLB. The QF-UFIDL encoding, however, associates with each propositional subformula a *formula predicate* that is a unary uninterpreted predicate $\boldsymbol{\theta} \in \mathcal{P}(\mathbb{Z})$. When subformula θ holds at instant i then $\boldsymbol{\theta}(i)$ holds. As the length of paths is fixed to $k+1$, and all paths start from 0, formula predicates are actually subsets of $\{0, \ldots, k+1\}$. Let θ be a propositional subformula of Φ, α, β be two a.t.t.'s and $\sim$ be any relation in DL; then the formula predicate associated with θ (denoted by the same name but written in bold face), is recursively defined as:

θ	$0 \leq i \leq k+1$
p	$\boldsymbol{\theta}(i) \Leftrightarrow p \in L(s_i)$
$\alpha \sim \beta$	$\boldsymbol{\theta}(i) \Leftrightarrow \boldsymbol{\alpha}(i) \sim \boldsymbol{\beta}(i)$
$\neg\phi$	$\boldsymbol{\theta}(i) \Leftrightarrow \neg\boldsymbol{\phi}(i)$
$\phi \wedge \psi$	$\boldsymbol{\theta}(i) \Leftrightarrow \boldsymbol{\phi}(i) \wedge \boldsymbol{\psi}(i)$

D. Encoding Temporal Operators

Temporal subformulae constraints define the basic temporal behavior of future and past operators, by using their usual fixpoint characterizations. Let ϕ and ψ be propositional subformulae of Φ, then:

θ	$0 \leq i \leq k$
$\mathbf{X}\phi$	$\boldsymbol{\theta}(i) \Leftrightarrow \boldsymbol{\phi}(i+1)$
$\phi\mathbf{U}\psi$	$\boldsymbol{\theta}(i) \Leftrightarrow \boldsymbol{\psi}(i) \vee (\boldsymbol{\phi}(i) \wedge \boldsymbol{\theta}(i+1))$
$\phi\mathbf{R}\psi$	$\boldsymbol{\theta}(i) \Leftrightarrow \boldsymbol{\psi}(i) \wedge (\boldsymbol{\phi}(i) \vee \boldsymbol{\theta}(i+1))$

To correctly define the semantic of past operators, the initial instant $i = 0$ has to be treated separately:

θ	$0 < i \leq k+1$	$i = 0$
$\mathbf{Y}\phi$	$\boldsymbol{\theta}(i) \Leftrightarrow \boldsymbol{\phi}(i-1)$	$\neg\boldsymbol{\theta}(0)$
$\mathbf{Z}\phi$	$\boldsymbol{\theta}(i) \Leftrightarrow \boldsymbol{\phi}(i-1)$	$\boldsymbol{\theta}(0)$
$\phi\mathbf{S}\psi$	$\boldsymbol{\theta}(i) \Leftrightarrow \boldsymbol{\psi}(i) \vee (\boldsymbol{\phi}(i) \wedge \boldsymbol{\theta}(i-1))$	$\boldsymbol{\theta}(0) \Leftrightarrow \boldsymbol{\psi}(0)$
$\phi\mathbf{T}\psi$	$\boldsymbol{\theta}(i) \Leftrightarrow \boldsymbol{\psi}(i) \wedge (\boldsymbol{\phi}(i) \vee \boldsymbol{\theta}(i-1))$	$\boldsymbol{\theta}(0) \Leftrightarrow \boldsymbol{\psi}(0)$

Last state constraints define an equivalence between truth in $k+1$ and those one indicated by $\boldsymbol{loop}$, since the instant $k+1$ is representative of the instant $\boldsymbol{loop}$ along periodic paths. Otherwise, truth values in $k+1$ are trivially false. These constraints have a similar structure to the corresponding Boolean ones, but here they are defined by only *one* DL constraint, for each subformula θ of Φ, w.r.t. the variable $\boldsymbol{loop}$:

$$\Big(\textstyle\bigwedge_{i=1}^{k}(\boldsymbol{loop} = i) \Rightarrow (\boldsymbol{\theta}(k+1) \Leftrightarrow \boldsymbol{\theta}(i))\Big) \wedge$$
$$\Big(\textstyle\bigwedge_{i=1}^{k} \neg(\boldsymbol{loop} = i) \Rightarrow (\neg\boldsymbol{\theta}(k+1))\Big).$$

Note that if a loop does not exist then the fixpoint semantics of $\mathbf{R}$ is exactly the one defined over finite acyclic paths in Section III. Finally, to correctly define the semantics of $\mathbf{U}$

and $\mathbf{R}$, their *eventuality* has to be accounted for. Briefly, if $\phi\mathbf{U}\psi$ holds at i, then ψ eventually holds in $j \geq i$; if $\phi\mathbf{R}\psi$ does not hold at i, then ψ eventually does not hold in $j \geq i$. Along finite paths of length k, eventualities must hold between 0 and k. If a loop exists, an eventuality may hold within the loop. The original Boolean encoding introduces k propositional variables for each $\phi\mathbf{U}\psi$ and $\phi\mathbf{R}\psi$ subformula of Φ, for all $1 \leq i \leq k$, which represent the eventuality of ψ implicit in the formula (see [10]). Instead, in the QF-UFIDL encoding, only *one* variable $\boldsymbol{j_\psi} \in \mathbb{Z}$ is introduced for each ψ occurring in a subformula $\phi\mathbf{U}\psi$ or $\phi\mathbf{R}\psi$; let $\boldsymbol{\ell}$ be a shorthand for the formula $\bigvee_{i=1}^{k}(\boldsymbol{loop} = i)$:

θ	
$\phi\mathbf{U}\psi$	$\boldsymbol{\ell} \Rightarrow (\boldsymbol{\theta}(k) \Rightarrow \boldsymbol{loop} \leq \boldsymbol{j_\psi} \leq k \wedge \boldsymbol{\psi}(\boldsymbol{j_\psi}))$
$\phi\mathbf{R}\psi$	$\boldsymbol{\ell} \Rightarrow (\neg\boldsymbol{\theta}(k) \Rightarrow \boldsymbol{loop} \leq \boldsymbol{j_\psi} \leq k \wedge \neg\boldsymbol{\psi}(\boldsymbol{j_\psi}))$

The complete encoding of Φ consists of the logical conjunction of all above constraints, together with Φ evaluated at the first instant of the time structure.

Let Φ be a PLTLB formula (i.e., purely propositional), then we can compare the dimension of the SAT-based encoding versus the QF-UFIDL one. If m is the total number of subformulae and n is the total number of temporal operators $\mathbf{U}$ and $\mathbf{R}$ occurring in Φ, then the SAT-based encoding requires $(2k+3)+(k+2)m+(k+1)n = O(k(m+n))$ fresh propositional variables. The QF-UFIDL encoding, instead, requires only $n+1$ integer variables ($\boldsymbol{loop}$ and $\boldsymbol{j_\psi}$) and m unary predicates (one for each subformula).

V. Case Study

To demonstrate our methodology, we use an example concerning two existing conversational services available on the Internet. These two services realize two lyric search engines. One is called *ChartLyrics*[1], the other *LyricWiki*[2].

ChartLyrics is a lyrics database sorted by artists or songs; the WSDL[3] of *ChartLyrics* provides three operations:

- *SearchLyric* to search available lyrics,
- *SearchLyricText* to search a song by means of some text within an available lyric text,
- *GetLyric* to retrieve the searched lyric.

LyricWiki is a free site where anyone can go to get reliable lyrics for any song from any artist. The WSDL of *LyricWiki*[4] provides several operations. Five of them are of interest for our purposes:

- *searchSongs* to search for a possible song on *LyricWiki* and get up to ten close matches,
- *checkSongExists* to check if a song exists in the *LyricWiki* database,
- *getSong* to get the lyrics for a searched *LyricWiki* song with the exact artist and song match,
- *searchArtists* to search for a possible artist by name and return up to ten close matches,
- *getArtist* to get the entire discography for a searched artist.

To get a lyric through *ChartLyrics*, a client can exploit the following sequence of operation invocations: *SearchLyric*, *GetLyric*. Conversely, to get a lyric through *LyricWiki*, a possible sequence of operation invocations is the following: *checkSongExists*, *searchSongs*, *getSong* (see the representation of the conversational protocols of *ChartLyrics* and *LyricWiki*, respectively, in Fig. 1 and Fig. 2).

If *LyricWiki* were part of a web application realized through a service composition, it could happen that, in certain circumstances, it would need to be replaced by *ChartLyrics* or by any other specialized search engine. This could happen, for instance, to accommodate the preferences of users having their preferred engine, or to handle the cases when *LyricWiki* is unavailable for any reason. The developer could code, by hand, the instructions to deal with any possible engine and its replacement. However, this approach does not allow the application to deal with search engines unknown at design time. A better solution, which would overcome this problem, is to build a mapping mechanism that dynamically handles the mismatches by automatically synthesizing a behavior protocol mapping script. The adaptation realized by the synthesized mapping script could state, e.g., that the sequence of *LyricWiki* operations *checkSongExists*, *searchSongs*, *getSong* maps to the sequence of *ChartLyrics* operations *SearchLyric*, *GetLyric*.

Let us consider as an example the expected service operation sequence *checkSongExists*, *searchSongs*, *getSong*, which brings the *LyricsWiki* behavior protocol automaton from state $start$ to state s_6 (see Fig. 2). We assume to have established a compatibility relation between services' data. Also, for the sake of brevity, the automata of Figures 1 and 2 are represented with this relation already established, though in practice this requires an additional mapping step (for more details see [6], [4]).

The automata describing service protocols and the expected service operation sequence are all formalized through suitable CLTLB(DL) formulae $expectedService$, $actualService$ and $expectedOperationSequence$, simply describing the transition relation between states. Then, we formulate the problem of checking if the expected service can be substituted by the actual service in terms of a bounded reachability problem over the automata describing the protocols of the expected and actual services. The problem consists of searching for a finite operation sequence on the actual service automaton which is substitutable to the expected operation sequence given as input. We check this condition by verifying that the actual service operation sequence should require no more input parameters than those provided to the expected service sequence, and it should provide at least the same parameters provided by

[1] http://www.chartlyrics.com/api.aspx
[2] http://lyrics.wikia.com/Main_Page
[3] http://api.chartlyrics.com/apiv1.asmx?WSDL
[4] http://lyrics.wikia.com/server.php?wsdl

the expected service sequence. To ensure this property we keep track, through instances of counters seen and needed (see Section II), of how many parameters of any given kind are provided as input to the expected service operations and of how many parameters of any given kind are returned by each actual service operation. For example, the behavior of counter needed is formalized by the CLTLB(DL) formula

$$\begin{aligned}
&\forall x\big(\,\mathrm{retdata}(x) \Rightarrow \\
&\quad (\mathrm{wait}(x) \wedge \mathrm{seen}(x) \geq 0 \Rightarrow \top \mathbf{U}\, \mathrm{resp}(x)) \wedge \\
&\quad (\neg\, \mathrm{wait}(x) \wedge \neg\, \mathrm{resp}(x) \Rightarrow \mathrm{needed}(x) = \mathbf{X}\, \mathrm{needed}(x)) \wedge \\
&\quad (\neg\, \mathrm{wait}(x) \wedge \mathrm{resp}(x) \Rightarrow \\
&\qquad\qquad \mathrm{needed}(x) = \mathbf{X}\, \mathrm{needed}(x) + 1) \wedge \\
&\quad (\mathrm{wait}(x) \Leftrightarrow \mathrm{needed}(x) = \mathbf{X}\, \mathrm{needed}(x) - 1)\big)
\end{aligned}$$

where $\mathrm{retdata}(x)$ holds if x is the type of an output parameter of a service (either expected or actual); $\mathrm{wait}(x)$ (resp. $\mathrm{resp}(x)$) holds when the expected (resp. actual) service is invoked and x is the type of one of the output parameters of the invoked service. For instance, when operation *checkSongExists* of *LyricWiki* is invoked, $wait(lyricsId)$ holds and when operation *SearchLyric* of *ChartLyrics* is invoked $resp(SongRank)$ holds. To completely state the reachability problem, an initial and a final condition over counters and states of automata are defined. The initial condition sets all counters to 0 and the states of the two LTSs to *start*. The final condition imposes $\forall x(\mathrm{retdata}(x) \Rightarrow \mathrm{needed}(x) \leq 0)$.

Finally, a solution for the bounded reachability problem can be obtained by checking the satisfiability of the conjunction of the CLTLB(DL) formulae mentioned above .

Considering the example sequence on *LyricsWiki*, a client expecting to invoke this sequence is assuming to provide as input to the first operation of the sequence a song and an artist. This will set the seen counter to 1 for both provided inputs. Moreover, it expects the invoked operation to return a *lyricsId* and a *lyricCheckSum*, which will increment the corresponding instances of the needed counter to 1. Considering the actual service protocol, our approach searches for an operation accepting a subset of the provided input data and providing a superset of the required return data.

The operation to be selected should leave the $start$ state as the state compatibility relation provided as input for the approach mandates the compatibility of state *start* of *LyricsWiki* with state *start* of *ChartLyrics*. In our example the invocation of *checkSongExists* makes *SearchLyric* the only suitable candidate. After the invocation of this actual service operation all instances of seen and those instances of needed associated to the output parameters of *checkSongExists* are reset to 0. The actual service operation returns also some extra data that are not required by the invoked expected service operation (i.e. song, artist, songRank, artistUrl, songUrl). In this case the reasoning mechanism offers two possible choices: extra data can be discarded (hence ignored also in the future), or they can be initially ignored, but stored for an eventual later use. The former strategy is more conservative, but it may also limit the possibility of the reasoning mechanism to find an adapter. The latter strategy may affect data consistency in some cases, as it allows using as a reply for an operation some data that have been received before the request has been actually issued, but it also opens the possibility of finding adapters in situations in which the former would fail. In this case study we use the latter strategy, hence the needed counters for those data that are not required as a response by the invoked expected service operation are set to -1.

After the invocation of *SearchLyric* the actual service goes in *SearchLyric_start* state. The next operation on the expected sequence to be invoked is *searchSongs*, which requires as input the names of the song to be searched and of its author and provides as return parameters the names of the artist and of the song, if they are found. Since the needed counters for both the name of the artist and of the song are set to -1, instances of those data have been previously stored, hence no operation shall be invoked on the actual service, which remains in state *SearchLyric_start*.

The last operation in the expected sequence is *getSong*, which requires as input artist and song names and the id and checkSum returned by the previously invoked *checkSongExists*. The expected service has again the same three operations of the previous step available, but this time there are two available candidates for selection: *searchSongs* and *GetLyric*. Given the data-flow constraints elicited before, *GetLyric* is the only available operation that can satisfy also the state compatibility relation. After the invocation of *GetLyric* the expected and actual services are in compatible states and the needed counter instances are all set to 0. Then, the actual service operation sequence found can be substituted to the expected service sequence.

A mapping script generated for the example sequence in this section is reported in Table I. Each step contains the state in which each one of the analyzed automata is, the operations in seq_{exp} and in seq_{act} that should be invoked in that step, and the exchanged data, if any. For each operation in seq_{exp} the adapter expects to receive an invocation for the expected service, and for each operation in seq_{act} the adapter performs an invocation to the actual service. The table shows also the updates for the seen and needed counters.

VI. Evaluation and Experimental Results

In order to evaluate the encoding presented in this paper we built a plug-in of Zot and we used it in three sets of experiments[5]:

- We created adapters for sequences of increasing length related to the case study presented in Section V. This set of experiments was used as a qualitative evaluation of the approach on examples taken from the real world.

[5]The experiments sets are available at http://home.dei.polimi.it/cavallaro/sefm10-experiments.html

Step	*Execution trace Content*	Counters value
1	*LyricWiki*State:start ; *LyricWiki*Operation:checkSongExists *LyricWiki*Input: song, artist; *LyricWiki*Output:lyricId, lyricCheckSum *chartLyrics*State:start; *LyricWiki*Operation:checkSongExists	All counters set to 0
2	*LyricWiki*State:s_1 *chartLyrics*Input: song, artist *chartLyrics*Output:song , artist, artistUrl, songRank, lyricsId, lyricChecksum *chartLyrics*State:start; *chartLyrics*Operation:searchLyric	seen(song) = seen(artist) = 1 needed(lyricId) = needed(lyricCheckSum) = 1
3	*LyricWiki*State:s_1; *LyricWiki*Operation:searchSongs *LyricWiki*Input:song, artist; *LyricWiki*Output:song, artist *chartLyrics*State:searchLyric_start	seen(song) = seen(artist) = 0 needed(lyricsId) = needed(lyricCheckSum) = 0 needed(artist) = needed(artistUrl) = -1 needed(song) = needed(songRank)= -1
4	*LyricWiki*State:s_5 *chartLyrics*State:searchLyric_start *chartLyrics*Operation: *None*	seen(song) = seen(artist) = 1 needed(song) = needed(artist) = 0
5	*LyricWiki*State:s_5; *LyricWiki*Operation: getSong *LyricWiki*Input: lyricId, song, lyricCheckSum, artist *LyricWiki*Output:song, artist, lyricCorrectUrl, Lyric *chartLyrics*State:searchLyric_start	No changes
6	*LyricWiki*State:s_6 *chartLyrics*Input: lyricId, lyricCheckSum *chartLyrics*Output: song, artist, artistUrl, lyricRank, Lyric, lyricCorrectUrl, lyricCoverArtUrl *chartLyrics*State:searchLyric_start *chartLyrics*Operation:getLyric	seen(song) = seen(artist) = 2 seen(lyricCheckSum) = seen(lyricId) = 1 needed(song) = needed(artist) = 1 needed(lyricCorrectUrl) = needed(Lyric) = 1
7	*LyricWiki*State:s_6 *LyricWiki*Operation: *None* *chartLyrics*State:end *chartLyrics*Operation: *None*	seen(lyricCheckSum) = seen(lyricId) = 0 needed(song) = needed(artist) = 0 needed(lyricCorrectUrl) = needed(Lyric) = 0 needed(artistUrl) = needed(lyricRank) = -1

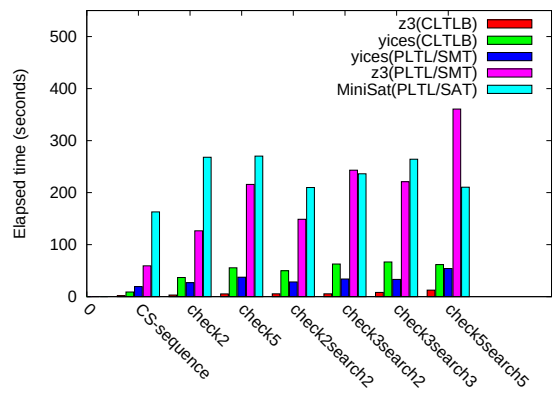
Table I
MAPPING SCRIPT GENERATED FOR THE CASE STUDY IN SECTION V

- We ran the same set of experiments on Zot using three different encodings, namely: the SAT-based encoding of PLTL (PLTL/SAT, see [13] for details); the SMT-based one of the same logic (PLTL/SMT, which is the one described in Section IV except that a.t.t.'s are encoded only for finite domains) and the SMT-based one of CLTLB(DL), featuring the encoding of the a.t.t.'s presented in Section IV-B. We measured elapsed time and occupied memory, and we compared the results to get an estimate of how the introduction of the SMT solver speeds up the adapter-building mechanism.
- We created some service interface models with growing number of parameters and tried to solve them with both the original version of the encoding and with the extensions. This set of experiments has the purpose to compare how much the new encoding scales on models larger than those found in common practice.

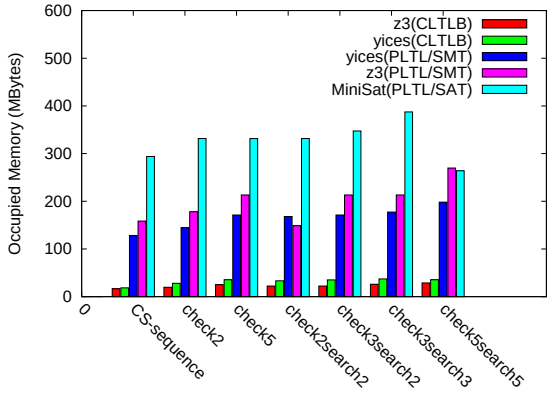
All experiments were run using the Common Lisp compiler SBCL 1.0.29.11 on a 2.50GHz Core2 Duo laptop with Linux and 4 GB RAM. The SMT solvers used in our tests are: Microsoft Z3 2.4 (http://research.microsoft.com/en-us/um/redmond/projects/z3/) and SRI Yices 2.0 prototype (http://yices.csl.sri.com/). For the SAT-based PLTL encoding we used MiniSat 2.0 beta (http://minisat.se/).

The first set of experiments was carried out selecting some operation sequences on the expected service presented in Section V. The selected sequences set comprises the simple sequence analyzed in the case study plus sequences of growing length obtained trying to execute up to 5 consecutive *searchSongs* and *checkSongExists* operations. We set the time bounds for the experiments using a simple heuristic, based on the sum of the states of the automata of the input services. In those cases in which the abstract sequence featured repeated invocations of the same operation, the time bound was augmented with the number of repetitions of each operation. This set of experiments produced a set of mapping scripts that we checked by inspection. Fig. 4(a) and Fig. 4(b) report the overall results. Fig. 4(b) shows that the CLTLB(DL) encoding has lower memory occupation than the SAT-based PLTL encoding for the same problem. Fig. 4(a) shows that the CLTLB(DL) encoding on Z3 performs much better than the others.

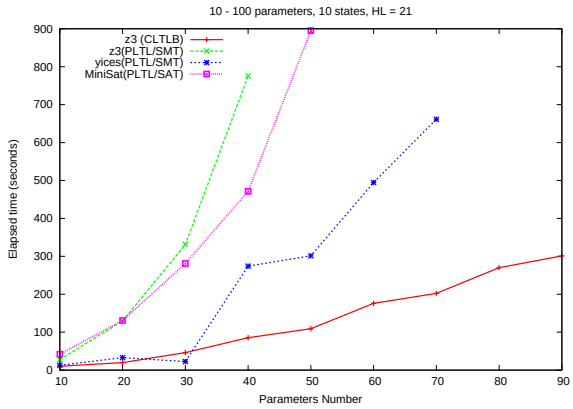
Lastly, we tried to push the limits of our technique to check its robustness. To do so, we generated simple service protocols featuring operations with a growing number of parameters. We chose this setting for our experiments based on our experience in the common practice, which suggests that services usually exhibit very simple protocols, while operations have sometimes a considerable number of parameters. Note that the models used in these experiments are much bigger than those commonly found in practice. The experiments are based on expected and actual services with 10 states, and a trace bound of 21 time instants. The results are shown in Fig. 4(c) and in Fig. 4(d). The number of parameters used in experiments ranges from 10 (i.e. each operation has 10 input and 10 output parameters) to 90. As shown in the figures, the CLTLB(DL) encoding on Z3 was the only one we managed to push up to 90 parameters, while we stopped experimenting much earlier with the PLTL encoding on Yices, Z3 and MiniSat. Note that in Fig. 4(c)-



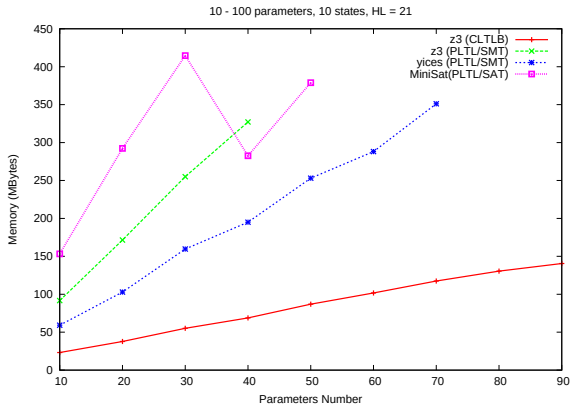
(a) Elapsed times on the second set of experiments



(b) Memory occupations on the second set of experiments



(c) Elapsed times on the third set of experiments



(d) Memory occupations on the third set of experiments

Figure 4. Experimental Results

4(d) the combination CLTLB(DL)/Yices is missing because of its poor performance on this set of experiments (the simplest case was solved in more than 500 seconds).

VII. Related Work

Our approach is closely related both to works supporting substitution of services and to works about verification using model checking. Many approaches that support the automatic generation of adapters (or equivalent mechanisms) are based on the use of ontologies and focus on non-conversational services (see for instance [6], [14]). They all assume that the usual WSDL definition of a service interface is enriched with some kinds of ontological annotations. At run-time, when a service bound to a composition needs to be substituted, a software agent generates a mapping by parsing such ontological annotations. *SCIROCO* [15] offers similar features but focuses on stateful services. It requires all services to be annotated with both a SAWSDL description and a WSResourceProperties [16] document, which represents the state of the service. When an invoked service becomes unavailable, *SCIROCO* exploits the SAWSDL annotations to find a set of candidates that expose a semantically matching interface. Then, the WS-ResourceProperties document associated with each candidate service is analyzed to find out if it is possible to bring the candidate in a state that is compatible with the state of the unavailable service. If this is possible, then this service is selected for replacement of the one that is unavailable. All these three approaches offer full run-time automation for service substitution, but as the services they consider are not conversational, they perform the mapping on a per-operation basis. An approach that generates adapters covering the case of interaction protocols mismatches is

presented in [17]. It assumes to start from a service composition and a service behavioral description both written in the BPEL language [18]. These are then translated in the *YAWL* formal language [19] and matched in order to identify an invocation trace in the service behavioral description that matches the one expected by the service composition. The matching algorithm is based on graph exploration and considers both control flow and data flow requirements. The approach presented in [20] offers similar features and has been implemented in the open source tool Dinapter (http://sourceforge.net/projects/dinapter). While both these approaches appear to fulfill our need for supporting interaction protocol mapping, they present some shortcoming in terms of performances, as shown in [3].

Although QF-UFIDL involves variables over infinite domain, our particular BMC of CLTLB(DL) formulae became effective because it is not used as an infinite-state model checking procedure. In general, transitions systems defined by arithmetic constraints provide a large class of infinite-state systems which are suitable for modeling a large variety of applications. So, intensive work has been devoted to identify useful classes with decidable reachability and safety properties [21], [22]. Some implemented procedures [23], [24] rely on a pure operational approach and the complexity of the decision problem of the underlying arithmetic (3-EXPTIME in the case of Presburger Logic) do not make them appropriate for runtime checking. Much effort is also devoted to study decidability and complexity of temporal logic of arithmetic constraints, [25], [26], [7], [8]. [27] proposes a semi-decision procedure aimed to be used for model checking of an extension of CTL* with Presburger constraints. Finally, an operational approach to BMC which exploits a direct translation of LTL formulae of arithmetic constraints is suggested in [28]. Our approach offers a mixed operational-descriptive BMC based on the satisfiability of CLTLB(DL) formulae which enjoys the NP-completeness of the decision problem of DL, significantly less than that of more complex theories.

VIII. Conclusion

In this paper we introduced an efficient encoding for a linear temporal logic with arithmetic constraints. Our encoding was found very suitable for application to a real problem taken from the SOA domain and showed better performances and lower memory occupation than the other encodings we compared it with. The research work is currently still ongoing. For future work we plan to further experiment with our encoding and to further investigate its theoretical properties.

Acknowledgments

Many thanks to E. Di Nitto, A. Morzenti, and P. San Pietro for the fruitful discussions and support. This research was partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom, and by the Italian Government, Project PRIN 2007 D-ASAP (2007XKEHFA).

References

[1] V. De Antonellis, M. Melchiori, L. De Santis, M. Mecella, E. Mussi, B. Pernici, and P. Plebani, "A layered architecture for flexible web service invocation," *Softw. Pract. Exper.*, vol. 36, no. 2, pp. 191–223, 2006.

[2] K. Verma, K. Gomadam, A. Sheth, J. Miller, and Z. Wu, "The METEOR-S approach for configuring and executing dynamic web processes," LSDIS Lab, University of Georgia, Athens, Georgia, Tech. Rep. LSDIS 05-001, 2005.

[3] L. Cavallaro, E. Di Nitto, and M. Pradella, "An automatic approach to enable replacement of conversational services," in *Proc. ICSOC/ServiceWave*, 2009, pp. 159–174.

[4] L. Cavallaro, E. Di Nitto, P. Pelliccione, M. Pradella, and M. Tivoli, "Synthesizing adapters for conversational web-services from their WSDL interface," in *Proc. SEAMS*, 2010.

[5] S. Ranise and C. Tinelli, "SMT-LIB 1.2," Tech. Rep., 2006, http://combination.cs.uiowa.edu/smtlib/.

[6] L. Cavallaro, G. Ripa, and M. Zuccalà, "Adapting service requests to actual service interfaces through semantic annotations," in *Proc. PESOS*, 2009.

[7] M. M. Bersani, A. Frigeri, M. Pradella, M. Rossi, A. Morzenti, and P. San Pietro, "Bounded Reachability for Temporal Logic over Constraint System," in *Proc. TIME*, 2010.

[8] H. Comon and V. Cortier, "Flatness Is Not a Weakness," in *Proc. CSL*, 2000, pp. 262–276.

[9] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 118–149, 2003.

[10] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan, "Linear encodings of bounded LTL model checking," *LMCS*, vol. 2, no. 5, 2006.

[11] M. Pradella, A. Morzenti, and P. San Pietro, "The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties," in *Proc. ESEC/SIGSOFT FSE*, 2007, pp. 312–320.

[12] ——, "A metric encoding for bounded model checking," in *Proc. FM*, ser. LNCS, vol. 5850. Springer, 2009, pp. 741–756.

[13] ——, "Refining real-time system specifications through bounded model- and satisfiability-checking," in *Proc. ASE*, 2008, pp. 119–127.

[14] C. Drumm, "Improving schema mapping by exploiting domain knowledge," Ph.D. dissertation, Universitat Karlsruhe, Fakultat fur Informatik, 2008.

[15] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras, "Dynamic service substitution in service-oriented architectures," in *Proc. SERVICES*, 2008, pp. 101–104.

[16] T. Schaeck and R. Thompson, "WS-ResourceProperties," http://docs.oasis-open.org/wsrp/Misc/, 2003.

[17] A. Brogi and R. Popescu, "Automated generation of BPEL adapters," in *Proceedings of ICSOC*, 2006, pp. 27–36.

[18] OASIS, "Web Services Business Process Execution Language Version 2.0," http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, 2007.

[19] W. M. P. van der Aalst and A. H. M. ter Hofstede, "YAWL: yet another workflow language," *Inf. Syst.*, vol. 30, no. 4, pp. 245–275, 2005.

[20] J. A. Martìn and E. Pimentel, "Automatic generation of adaptation contracts," in *Proc. FOCLASA*, 2008.

[21] L. Fribourg and H. Olsén, "Proving Safety Properties of Infinite State Systems by Compilation into Presburger Arithmetic," in *Proc. CONCUR*, 1997, pp. 213–227.

[22] H. Comon and Y. Jurski, "Multiple Counters Automata, Safety Analysis and Presburger Arithmetic," in *Proc. CAV*, 1998, pp. 268–279.

[23] B. Boigelot, "Symbolic methods for exploring infinite state spaces," Ph.D. dissertation, Université de Liège, 1998.

[24] A. Annichini, A. Bouajjani, and M. Sighireanu, "TReX: A Tool for Reachability Analysis of Complex Systems," in *Proc. CAV*, 2001, pp. 368–372.

[25] S. Demri, "LTL over Integer Periodicity Constraints: (Extended Abstract)," in *Proc. FoSSaCS*, 2004, pp. 121–135.

[26] S. Demri and D. D'Souza, "An automata-theoretic approach to constraint LTL," in *Proc. FSTTCS*, 2002, pp. 121–132.

[27] S. Demri, A. Finkel, V. Goranko, and G. van Drimmelen, "Towards a Model-Checker for Counter Systems," in *Proc. ATVA*, 2006, pp. 493–507.

[28] L. M. de Moura, H. Rueß, and M. Sorea, "Lazy theorem proving for bounded model checking over infinite domains," in *Proc. CADE*, 2002, pp. 438–455.