Keep It Small, Keep It Real: Efficient Run-Time Verification of Web Service Compositions

Luciano Baresi[1], Domenico Bianculli[2], Sam Guinea[1], and Paola Spoletini[3]

[1] Politecnico di Milano - DEEP-SE Group - DEI, Milano, Italy
{luciano.baresi, sam.guinea}@polimi.it
[2] University of Lugano - Faculty of Informatics, Lugano, Switzerland
domenico.bianculli@lu.unisi.ch
[3] Università dell'Insubria - DSCPI, Como, Italy
paola.spoletini@uninsubria.it

Abstract. Service compositions leverage remote services to deliver added-value distributed applications. Since services are administered and run by independent parties, the governance of service compositions is intrinsically decentralized and services may evolve independently over time. In this context, pre-deployment verification can only provide limited guarantees, while continuous run-time verification is needed to probe and guarantee the correctness of compositions at run time.

This paper addresses the issue of efficiency in the run-time verification of service compositions described in BPEL. It considers an existing monitoring approach based on ALBERT, which is a temporal logic language suitable for asserting both functional and non-functional properties, and shows how to obtain the efficient run-time verification of ALBERT formulae. The paper introduces an operational semantics for ALBERT through an extension of alternating automata, and explains how to optimize it to produce smarter, and thus more efficient, encodings of defined formulae. The optimized operational semantics can then be the basis for an efficient implementation of the run-time verification framework.

1 Introduction

Services represent reusable software components that provide their functionality to many clients through a standardized network and middleware infrastructure. Clients may combine services in different ways, to create new composite applications that can be themselves published as a service. In the realm of Web services, service compositions are usually described by means of the BPEL [1] language, which supports the definition of workflow-like service compositions.

BPEL orchestrations usually involve multiple stakeholders, as service aggregators rely on parts that are owned and managed by other organizations. The overall quality of a BPEL process largely depends on the quality of the composed services. Since these services are run and administered autonomously, in a decentralized manner, providers are entitled to change them freely. For this reason, the actual partner services invoked by a composite service can evolve (or even change) at run time. Pre-deployment verification is of limited usefulness; run-time verification becomes mandatory to probe and check the quality and correctness of service compositions while they execute.

Run-time verification may check different properties, ranging from quality of service parameters (e.g., response time, throughput) to behavioral assertions. These properties are often expressed by means of special-purpose languages. In [2], we introduced ALBERT, an assertion language for the specification of functional and non-functional temporal properties of BPEL processes. ALBERT plays a key-role in SAVVY-WS [3], our proposal for an integrated design- and run-time verification methodology.

This paper focuses on the *efficient* verification of ALBERT formulae at run time. It starts by proposing an operational semantics for ALBERT based on the correlation between temporal logic and a class of alternating automata, called ASA (ALBERT's Semantics Automata). Since this "plain" operational model would lead to quite inefficient verifications, the paper also proposes a smart encoding of ALBERT formulae by means of an optimized semantics defined in terms of an extension of ASA, called LASA (Limited ASA). This new operational semantics is equivalent to the previous one, but fosters more efficient verifications. Experimental results corroborate this hypothesis and show how the proposed optimization limits the number of threads needed for a complete evaluation of a given formula.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to ALBERT. Section 3 presents the "plain" semantics ascribed to ALBERT in terms of our extension of alternating automata. Section 4 explains how to optimize the mapping of ALBERT formulae onto the formal model, and Sect. 5 fosters this hypothesis by means of some experimental results. Finally, Sect. 6 surveys related work, and Sect. 7 concludes the paper.

2 ALBERT in a Nutshell

The aim of this section is to accustom the reader with ALBERT, focusing on the main aspects that are needed to understand the theoretical framework presented in the paper.

ALBERT [2] is a temporal assertion language for stating functional and non-functional properties of BPEL workflows. ALBERT formulae predicate over *internal* and *external* variables. The former consist of data pertaining to the internal state of the BPEL process in execution. The latter are data that are useful for the specification, but are not part of the process' business logic and must be obtained externally (e.g., the values returned by some external services).

Given a finite set of variables V and a finite set of natural constants C, an ALBERT formula ϕ is defined by the following syntax:

$$\begin{array}{l}\phi ::= \chi \quad | \quad \neg\phi \quad | \quad \phi \wedge \phi \quad | \quad (\ \mathsf{op\ id\ in\ var}\ ;\ \phi\ ) \quad | \\ \mathit{Becomes}(\chi) \quad | \quad \mathit{Until}(\phi,\phi) \quad | \quad \mathit{Between}(\phi,\phi,K) \quad | \\ \mathit{Within}(\phi,K) \quad | \quad \mathit{InFuture}(\phi,K)\end{array}$$

$$
\begin{aligned}
&\chi ::= \psi\ \mathsf{relop}\ \psi \quad | \quad \neg\chi \quad | \quad \chi \wedge \chi \quad | \quad \mathit{onEvent}(\mu)\\
&\psi ::= \mathsf{var} \quad | \quad \psi\ \mathsf{arop}\ \psi \quad | \quad \mathsf{const} \quad | \quad \mathit{past}(\psi, \mathit{onEvent}(\mu), n) \quad |\\
&\mathit{count}(\chi, K) \quad | \quad \mathsf{fun}(\psi, K) \quad | \quad \mathit{elapsed}(\mathit{onEvent}(\mu))\\
&\mathsf{op} ::= \texttt{forall} \quad | \quad \texttt{exists}\\
&\mathsf{relop} ::= < \quad | \quad \leq \quad | \quad = \quad | \quad \geq \quad | \quad >\\
&\mathsf{arop} ::= + \quad | \quad - \quad | \quad \times \quad | \quad \div\\
&\mathsf{fun} ::= \mathit{sum} \quad | \quad \mathit{avg} \quad | \quad \mathit{min} \quad | \quad \mathit{max}
\end{aligned}
$$

where $\mathsf{var} \in V$, $\mathsf{const} \in C$, $n \in \mathbb{N}$, $K \in \mathbb{R}^+$ and *onEvent* is an event predicate. *Becomes*, *Until*, *Between*, *Within* and *InFuture* are temporal operators. *count*, *elapsed*, *past*, and all the functions derivable from the non-terminal fun are temporal functions of the language. Parameter μ denotes an event: it may identify the *start* or the *end* of an *invoke*, *reply* or *receive* activity, the reception of a message by a *pick* or an *event handler*, or the execution of any other BPEL activity. The above syntax only defines the language's core constructs. The usual logical derivations are used to define other connectives and temporal operators (e.g., $\vee$, *Always*, *Eventually*).

The informal meaning of ALBERT formulae can be explained by referring to sequences of states of the BPEL process, each of which represents a snapshot of the variables of the process, taken at a certain time instant, when the process is executing a certain set of activities.

Sequences of process states are strictly monotonic with respect to time. Between successive states there is always at least one time-consuming interaction with the outside world or the execution of an internal BPEL activity (e.g., an *assign* activity) or the occurrence of an event.

All ALBERT formulae represent invariant assertions over a BPEL process, therefore they are understood to be in the scope of an implicit universal temporal quantification, i.e., each formula is prefixed by an *Always* temporal operator. The predicate *onEvent* can be used to express a formula that must hold when the execution reaches a given point of the workflow. In the case where the parameter μ denotes *assign*, *pick*, *event handler*, or the *end* of *invoke*, *reply* or *receive* activities, *onEvent* is true in a state whose label identifies the corresponding activity. In the case of the *start* of an *invoke*, *reply* or *receive* activity, it is true in a state if the label of the next state in the sequence identifies the corresponding activity. In the case of a *while* or a *switch* activity, it is true in the state where the condition is evaluated.

Function $\mathit{past}(\psi, \mathit{onEvent}(\mu), n)$ returns the value of ψ in the nth past state in which $\mathit{onEvent}(\mu)$ is true. Function $\mathit{count}(\chi, K)$, evaluated in a state whose time-stamp is t_j, computes the number of states in which χ is true, and whose time-stamp is greater than or equal to $t_j - K$. The non-terminal fun stands for any function (e.g., average, sum, minimum, maximum ...) that can be applied to sets of numerical values. The function, evaluated in a state whose time-stamp is t_j, is applied to all values of expression ψ in all states whose time-stamp is greater than or equal to $t_j - K$. Function $\mathit{elapsed}(\mathit{onEvent}(\mu))$, evaluated in a state whose time-stamp is t_j, returns the difference between t_j and the time-stamp of the most recent past state in which $\mathit{onEvent}(\mu)$ is true. Since these functions compute their value from a trace storing a past history of states, their value becomes part of the process state. Moreover, a change in the value of function *count* and of the functions derivable from non-terminal fun may lead to the generation of a new state.

Temporal predicate *Becomes* is evaluated on two adjacent elements of the sequence of states. The formula is true if its argument is true in the current state, and false in the previous. The temporal predicate $Until(\phi_1, \phi_2)$ is true in a given state if ϕ_2 is true in the current state, or eventually in a future state, and ϕ_1 holds in all the states from the current (included) until that state (excluded). The temporal predicate $Between(\phi_1, \phi_2, K)$ is true in a given state if both ϕ_1 and ϕ_2 will be eventually true, with ϕ_2 occurring exactly after K time instants from the first time in which ϕ_1 was true.The temporal predicate $Within(\phi, K)$ is true in a given state if ϕ is true at most after K time instants. Predicate $InFuture(\phi, K)$ is true in a given state if ϕ is true in exactly K time instants.

Finally, boolean, relational, and arithmetic operators have the conventional meaning; the same is true for quantifiers.

3 ALBERT's Operational Semantics

The sequence of process states is linear and analogous to the sequence of states on which the operators of Linear Time Logic (LTL) — either in its classical definition or in the one with both modalities — are evaluated. The only main difference is that ALBERT operators also contain an explicit reference to time-stamps. Therefore, ALBERT temporal predicates can be described in terms of LTL operators. Furthermore, we consider sequences of infinite length since *a priori* we suppose that Web service compositions can be involved in long-running, never-ending business transactions. Notice that this does not represent a limitation if the system is stopped: in that case, the formulae are evaluated on an infinite sequence comprised of a prefix, represented by the states collected until that moment, and of a suffix of the form $false^{\omega}$.

Let s_c be the current state in a sequence of states and s_i be a sequent state that is at most K time instants after s_c and such that the successor state s_{i+1} comes more than K time instants after s_c, i.e., with the reference to time-stamps, $t_i - t_c \leq K$ and $t_{i+1} - t_c > K$. An ALBERT temporal predicate can be evaluated in state s_c according to the following equivalences with formulae of LTL with both modalities[4]:

- $Becomes(\chi) \equiv \mathsf{Y}(\neg\chi) \wedge \chi$
- $Until(\phi_1, \phi_2) \equiv \phi_1 \mathsf{U} \phi_2$
- $InFuture(\phi, K) \equiv \mathsf{X}^i(\phi)$
- $Within(\phi, K) \equiv \phi \vee \mathsf{X}(\phi) \vee \ldots \vee \mathsf{X}^{i-1} \vee \mathsf{X}^i(\phi)$

The temporal predicate $Between(\phi_1, \phi_2, K)$ is derived and can be expressed as $(\neg\phi_1)\mathsf{U}(\phi_1 \wedge InFuture(\phi_2, K))$.

[4] Y stands for "Yesterday", U for "Until", X for "neXt" and X^{i} for X nested i times.

Since ALBERT can be described in terms of LTL operators, in which however the number of nested Xs is not known *a priori*, we can exploit the well-known correlation between temporal logic and Büchi alternating automata (BAA) [4], as presented in [5], to give ALBERT an operational semantics that could be implemented straightforwardly. BAA generalize the traditional concept of non-determinism by supporting both *existential* and *universal* non-deterministic branching.

Since ALBERT's temporal model involves both a sequence of states and operations on their time-stamps, in the rest of this section we first introduce ALBERT's Semantics Automata (ASA), an extension of BAA that uses variables to deal with time-stamps, and then we show how the classical correlation between BAA and LTL can be reformulated for ASA and ALBERT.

3.1 ALBERT's Semantics Automata

Informally speaking, a BAA is a finite state automaton that recognizes words of infinite length and supports two branching modalities, universal and existential. These modalities are formally expressed in the model through positive Boolean combinations of formulae; given a set M of propositions, $\mathcal{B}^+(M)$ denotes the set of positive Boolean formulae over M built from elements in M using $\wedge$ and $\vee$ but not $\neg$, plus the formulae *true* and *false*. Universal branching in a BAA potentially allows for reducing the dimension of the automaton in which parallelism is not made explicit at design time.

Dealing with ALBERT formulae in a concise way requires that the BAA model be enriched with a set of bounded time counters, and with the corresponding assignment and comparison operators, to take care of the explicit temporal aspects. Formally, given a finite set $C_K = \{v_1, \ldots, v_n\}$ of time counters ranging over the non-negative rational numbers $\mathbb{Q}^+$ and bounded in value by a positive integer K, let Ψ_{C_K} be the set of counter constraints of the form $v \boxdot c$ where $v \in C_K$, $\boxdot \in \{<, \leq, =, \neq, >, \geq\}$ and $c \in \mathbb{Q}^+$. For the same set C_K, let Υ_{C_K} be the set of assignments over C_K of the form $v \leftarrow c$, where $v \in C_K$ and $c \in \mathbb{Q}^+$ and $c \leq K$, including also the empty assignment ε_Υ. ALBERT's Semantics Automata are defined as follows.

Definition 1. *An ASA is a tuple* $(\Sigma, Q, C_K, q_0, \delta, F)$ *where* Σ *is a finite alphabet,* Q *is a set of states,* C_K *is a finite set of time counters bounded in value by a positive integer* K*,* $q_0 \in Q$ *is the initial state,* $\delta : Q \times \wp(\Sigma) \times \mathbb{Q}^+ \times \Psi_{C_K} \rightarrow \mathcal{B}^+(Q \times \Upsilon_{C_K})$ *is the transition function and* $F \subseteq Q$ *is a set of accepting states.*

For the sake of readability, when indicating the elements in $\mathcal{B}^+(Q \times \Upsilon_{C_K})$ we will use the symbol / to separate the component in Q from the component in Υ_{C_K}.

An ASA accepts (or rejects) timed ω-words that are defined as sequences $w = w_1 w_2 \ldots = (a_1, t_1)(a_2, t_2) \ldots$ of pairs from $\wp(\Sigma) \times \mathbb{Q}^+$. For each $i > 1$, t_i describes the amount of time passed between reading a_{i-1} and a_i and t_1 represents the amount of time passed from the initial time (0) to the instant when a_1 was read. We also define the functions $D(w_i)$ and $t(w_i)$ that project, respectively, the data and the time component of the ith symbol of a word w. Due to universal branching, BAA's (and consequently ASA's) runs are not sequences, but trees. Indeed, every time a universal branch is taken, the automaton goes in all the states expressed by the $\wedge$ combination of formulae; hence, more than one state can be reached at the same time, as in a tree structure. This can be seen as the process of creating a duplicate of the automaton, at a certain level of the tree, for each state reached when performing the universal branch. A run of an ASA is *accepting* if every path starting from the root of the tree (corresponding to the run) hits accepting states infinitely often.

3.2 From ALBERT to ASA

Our proposal is to use an ASA for the run-time verification of an ALBERT formula, by defining the semantics of ALBERT formulae in terms of the operational model represented by the class of ASA. The implementation of the run-time checker becomes straightforward, as it follows the definition of the operational model. Indeed, while the truth value of a formula depends on the word on which it is evaluated, the equivalent corresponding automaton accepts the same word if and only if the formula is true on the word. Moreover, as ALBERT formulae represent invariant assertions over a BPEL process, the automaton equivalent to the formula to be verified, is supposed to run until the BPEL process for which ALBERT formulae are defined, is executed.

The basic idea is that an ASA equivalent to an ALBERT formula can be built from the latter (in the same way as a BAA can be derived from an LTL formula) by constructing a state for each temporal sub-formula in the formula, and by defining the transition relation between pairs of states $\langle q_j, q_k \rangle$ only if the truth value of the formula represented by state q_j depends on the truth value of the formula represented by state q_k. Moreover, the boolean connectors $\wedge$ and $\vee$ are implicitly represented by means of universal and existential branching.

In the following definition of the semantics, we do not consider ALBERT functions, but we treat them as part of the process state, as described in Sect. 2.

Standard Semantics. Let ϕ be an ALBERT formula, X the finite set of atomic propositions that occur in ϕ, and $\mathit{Sf}(\phi)$ the set of sub-formulae of ϕ. In order to define the semantics, we introduce some further definitions. Given an ALBERT formula ϕ, $\mathit{Dual}(\phi)$ is a formula obtained by interchanging in ϕ *true* and *false*, $\wedge$ and $\vee$, and complementing all the sub-formulae of ϕ. Moreover, let $H_D : \mathbb{N} \rightarrow \wp(X)$ and $H_t : \mathbb{N} \rightarrow \mathbb{Q}^+$ be, respectively the data[5] and the time history functions, which return, for a given n, respectively, the subset of atomic propositions that held in, and the time-stamp of, the nth-last data collection performed by the run-time checker. The ASA for ϕ is a tuple $(\Sigma,\ Q,\ C_K,\ q_0,\ \delta,\ F)$ where $\Sigma = X$, C_K is a finite set of time bounded counters such that $|C_K| \leq |\mathit{Sf}(\phi)|$, $Q = \{\gamma \mid \gamma \in \mathit{Sf}(\phi) \text{ or } \neg\gamma \in \mathit{Sf}(\phi)\}$, K is the greatest bounded temporal distance occurring in the temporal predicates of ϕ, $q_0 = \phi$, and $F = \{\gamma \mid \gamma \in Q \text{ and } \gamma \text{ has the form } \neg\, \mathit{Until}(\phi_1, \phi_2)\}$. The transition function δ is defined as follows, where χ, ϕ_1, ϕ_2

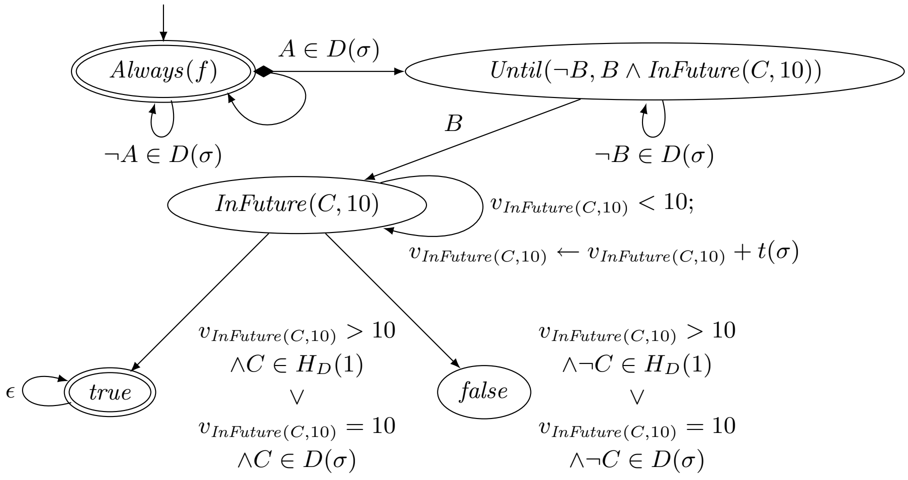
[5] Notice that $\forall \sigma, D(\sigma) = H_D(0)$.

are ALBERT (sub)formulae, σ is an input symbol, which is actually a pair from $\wp(X) \times \mathbb{Q}^+$, $v_\Psi \in \Psi_{C_K}$ is a generic constraint on a counter $v \in C_K$ and $v_\Upsilon \in \Upsilon_{C_K}$ is a generic assignment to a counter $v \in C_K$:

- $\delta(\chi, \sigma, v_\Psi) = \mathit{true}/v \leftarrow 0$ if $\chi \neq \mathit{onEvent}(\mu)$ where μ is a *start* event and $\chi \in D(\sigma)$;
- $\delta(\chi, \sigma, v_\Psi) = \mathit{false}/v \leftarrow 0$ if $\chi \neq \mathit{onEvent}(\mu)$ where μ is a *start* event and $\chi \notin D(\sigma)$;
- $\delta(\phi_1 \wedge \phi_2, \sigma, v_\Psi) = \delta(\phi_1, \sigma, v_\Psi) \wedge \delta(\phi_2, \sigma, v_\Psi)$;
- $\delta(\neg\phi, \sigma, v_\Psi) = \delta(\mathit{Dual}(\phi), \sigma, v_\Psi)$;
- $\delta(\mathit{Becomes}(\chi), \sigma, v_\Psi) = \mathit{true}/v \leftarrow 0$ if $\chi \neq \mathit{onEvent}(\mu)$ where μ is a *start* event, $\chi \in D(\sigma)$, and $\chi \notin H_D(1)$;
- $\delta(\mathit{Becomes}(\chi), \sigma, v_\Psi) = \mathit{false}/v \leftarrow 0$ if $\chi \neq \mathit{onEvent}(\mu)$ where μ is a *start* event and $\chi \notin D(\sigma)$ or $\chi \in H_D(1)$;
- $\delta(\mathit{InFuture}(\phi, K), \sigma, v_{\mathit{InFuture}(\phi,K)} = J) = \mathit{InFuture}(\phi, K)/v_{\mathit{InFuture}(\phi,K)} \leftarrow (J + t(\sigma))$ if $J < K$;
- $\delta(\mathit{InFuture}(\phi, K), \sigma, v_{\mathit{InFuture}(\phi,K)} = J) = \phi/v_{\mathit{InFuture}(\phi,K)} \leftarrow 0$ if $J = K$;
- $\delta(\mathit{InFuture}(\phi, K), \sigma, v_{\mathit{InFuture}(\phi,K)} = J) = \mathit{Previous}(\phi, v_{\mathit{InFuture}(\phi,K)} = J)/ v_{\mathit{InFuture}(\phi,K)} \leftarrow 0$ if $J > K$, where $\mathit{Previous}(\phi, v_\Psi)$ is equal to the Q component returned by $\delta(\phi, (H_D(1), H_t(1)), v_\Psi)$;
- $\delta(\mathit{Until}(\phi_1, \phi_2), \sigma, v_\Psi) = \delta(\phi_2, \sigma, v_\Psi) \vee (\delta(\phi_1, \sigma, v_\Psi) \wedge \mathit{Until}(\phi_1, \phi_2)/\varepsilon_\Upsilon)$;
- $\delta(\mathit{Within}(\phi, K), \sigma, v_{\mathit{Within}(\phi,K)} = J) = \mathit{false}/v_{\mathit{Within}(\phi,K)} \leftarrow 0$ if $J > K$;
- $\delta(\mathit{Within}(\phi, K), \sigma, v_{\mathit{Within}(\phi,K)} = J) = \delta(\phi, v_{\mathit{Within}(\phi,K)} = J) \vee \mathit{Within}(\phi, K)/ v_{\mathit{Within}(\phi,K)} \leftarrow (J + t(\sigma))$ if $J \leq K$.

For the sake of conciseness, in the above definition we omitted the semantics of: (a) the temporal operator $\mathit{Between}(\phi_1, \phi_2, K)$, since it is equivalent to the formula $\mathit{Until}(\neg\phi_1, \phi_1 \wedge \mathit{InFuture}(\phi_2, K))$; (b) (sub)formulae of the form $\chi = \mathit{onEvent}(\mu)$, where μ is a *start* event, since its semantics is equivalent to the one of the formula $\mathit{InFuture}(\chi, 1)$.

Figure 1 illustrates the ASA for the invariant $f \equiv A \implies \mathit{Between}(B, C, 10)$, which is equivalent, as a result of the logic equivalences mentioned above, to the formula $\neg A \vee \mathit{Until}(\neg B, B \wedge \mathit{InFuture}(C, 10))$. This ASA can be systematically derived from the definition of the operational semantics. The number of states in the resulting ASA is equal to the number of temporal operators plus two states for representing acceptance and rejection. In this case we have five states: one for the formula itself (since it is an invariant, the formula is implicitly contained within an *Always*), two for the *Until* and *InFuture* operators, one for acceptance, and one for rejection. The transition relation of the state containing the complete formula states that the automaton must stay in an acceptance state as long as $\neg A$ is true, i.e., an A is not received. On the other hand, as soon as A is received, the automaton must both stay in the same state to continue to check for A (due to the implicit *Always*), and move to another state to check $\mathit{Between}(B, C, 10)$ — transformed to $\mathit{Until}(\neg B, B \wedge \mathit{InFuture}(C, 10))$ — by creating a new copy of the automaton (the conjunction of the two copies is represented by a ◆). This copy remains in that state until a B is received. When this occurs, it moves to yet another state that checks the value of C while keeping an eye on the time counter $v_{\mathit{InFuture}(C,10)}$.



$Always(f)$
$A \in D(\sigma)$
$Until(\neg B, B \wedge InFuture(C, 10))$
$\neg A \in D(\sigma)$
B
$\neg B \in D(\sigma)$
$InFuture(C, 10)$
$v_{InFuture(C,10)} < 10;$
$v_{InFuture(C,10)} \leftarrow v_{InFuture(C,10)} + t(\sigma)$
$v_{InFuture(C,10)} > 10$
$\wedge C \in H_D(1)$
$\vee$
$v_{InFuture(C,10)} = 10$
$\wedge C \in D(\sigma)$
ϵ
$true$
$false$
$v_{InFuture(C,10)} > 10$
$\wedge \neg C \in H_D(1)$
$\vee$
$v_{InFuture(C,10)} = 10$
$\wedge \neg C \in D(\sigma)$

Fig. 1. ASA equivalent to the ALBERT invariant $f \equiv A \implies Between(B, C, 10)$

4 Towards an Efficient Implementation

The advantage of using alternating automata is that through *universal branching* we do not need to explicitly represent parallelism. The representation is exponentially more concise than standard Büchi automata. Universal branching, however, leads to the activation of multiple copies of the automaton. A direct implementation would spawn a new thread for each duplicated copy — an obvious efficiency bottleneck of the approach. For example, the evaluation of the ALBERT formula in Fig. 1, shows that when the automaton is in the initial state, it duplicates whenever an A is received. This is a problem because, if the automaton continues to duplicate (i.e., A is true in each state) without ever receiving B, the number of copies that are created can be unbounded.

This highlights the need to optimize the approach by fine-tuning the theoretical foundation with respect to implementation needs. An unbounded number of automaton duplications is unacceptable, and even a bounded but continuous duplication can be quite inefficient. This is why we propose to limit the number of duplications, while preserving the correspondence between the automaton and the ALBERT formula.

In the following, we informally describe how our optimization works, on a per-operator case. Since our ASA run on infinite words, when we use the verb "terminate", we refer to the situation in which (a copy of) an automaton reaches the *true* state, reports that the ALBERT property has not been violated, and remains in that state. Moreover, we assume that our run-time verification framework supports two modes of operation, which differ from each other in the way the system behaves when a violation of a property is detected. In *standard* mode the system logs the violation and continues the execution of the process; in *crit-*

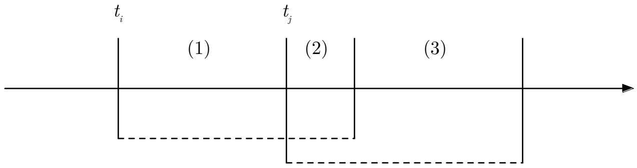


Fig. 2. Time line with overlapping regions of different activations of a *Within* operator

ical mode the framework stops the process execution, so that the cause of the violation can be dealt with immediately.

The evaluation of formulae of the form *Until*(ϕ_1, ϕ_2) could lead to an unbounded number of duplications. Indeed, every time the corresponding automaton is activated, it checks if ϕ_2 holds. If it does, the automaton terminates in the *true* state. If both ϕ_1 and ϕ_2 do not hold, it terminates in the *false* state. Finally, if ϕ_2 does not hold but ϕ_1 does, it continuously checks ϕ_1, waiting to terminate when ϕ_2 becomes true or ϕ_1 becomes false. If the same formula has to be checked again, a new copy of the automaton would be required. If a preexisting copy is still active, it will be checking ϕ_1, which is the exact behavior required of the new copy. Therefore, only one copy of the automaton is needed to evaluate the formula. This avoids a potentially infinite number of duplicates, which could occur when ϕ_1 is always true and ϕ_2 never becomes true. Since the *Between* can be seen as an *Until*, it benefits from the above considerations too.

As for the evaluation of formulae of the form *Within*(ϕ, K), the duplication is bounded by the number of states that occur, in the sequence of process states, in an interval of K time instances. This is true if the run-time verification framework operates in *standard* mode. However, if the operation mode is *critical*, e.g., if the discovery of a violation leads to a complete halt in the verification activities, we can use just one duplicate. Consider, for example, the time-line sketched in Fig. 2, where two copies of the automaton corresponding to the sub-formula *Within*(ϕ, K) are activated, one at time instant t_i and the other at instant t_j, with $t_i < t_j < t_i + K$. If ϕ evaluates to true in region (1), the first duplicate terminates in the *true* state, meaning the second duplicate activated in t_j is actually the only copy running. If ϕ evaluates to true in region (2), it will make both copies terminate in the *true* state, meaning the second duplicate is actually not needed. Finally, if ϕ is true for the first time in region (3), it implicitly induces a violation of the property, which is detected by the first copy of the automaton. When the automaton notifies the violation, if the framework is operating in *critical* mode, the verification activities are stopped, meaning, once again, that the second copy of the automaton is not needed.

4.1 A Formal Model for an Efficient Implementation

We can now formalize the above intuitions by defining a variation of ASA for which an efficient implementation can be derived, and showing its equivalence to the original ASA, in terms of which the standard semantics was defined.

We first need to modify the ASA in a way that leads to a limited number of duplications. The model is abbreviated as LASA, which stands for Limited ASA.

Given a set $B = \{f_1, \ldots, f_n\}$ of boolean variables, Ψ_B^{Dup} is the set of constraints over B, of the form $f_i = 0$ or $f_i = 1$ with $f_i \in B$, including the no constraint $\varepsilon_{\Psi^{Dup}}$, and Υ_B^{As} is the set of assignments over B of the form $f_i \leftarrow 0$ or $f_i \leftarrow 1$ with $f_i \in B$.

Definition 2. *A LASA is a tuple $(\Sigma, Q, C_K, q_0, \delta', B, F)$ where Σ, Q, C_K, q_0, F are defined as in an ASA, B is a set of boolean variables and the transition function δ' is defined as $\delta' : Q \times \wp(\Sigma) \times \mathbb{Q}^+ \times \Psi_{C_K} \times \Psi_B^{Dup} \times \Upsilon_B^{As} \rightarrow \mathcal{B}^+(Q \times \Upsilon_{C_K} \times \Upsilon_B^{As})$. The acceptance condition of a LASA is defined similarly to the one of an ASA.*

When a LASA is used to model an ALBERT formula ϕ, $|B| \leq |\mathit{Sf}(\phi)|$. The variables in B allow us to keep track of duplications. Non-universal transitions are not involved in duplications and therefore are not needed. If a flag f_i is set to 0, there are currently no active duplicates for the state reached by the transition. We use assignments in Υ_B^{As} to change the values of flags when a duplicate is created, to disable the transition. The value of a flag can be changed back to 0 when the corresponding duplicate terminates. Notice that variables in B are initially set to 0 and changed according to the Υ_B^{As} component in the transition function.

Optimized Semantics 1 (*standard* mode). Let ϕ be an ALBERT formula, and let $B = \{f_1, \ldots, f_n\}$ be the set containing a boolean variable for each element in $\mathit{Sf}(\phi)$. The transition function δ' of the LASA for ϕ is defined as follows, by redefining[6] the original δ of an ASA, where $f_\Upsilon \in \Upsilon_B^{As}$ is a generic assignment on a variable $f_i \in B$:

- $\delta'(\mathit{Until}(\phi_1, \phi_2), \sigma, v_\Psi, f_{\mathit{Until}(\phi_1,\phi_2)} = 0, f_\Upsilon) =$ $\delta'(\phi_2, \sigma, v_\Psi, \varepsilon_{\Psi^{Dup}}, f_{\mathit{Until}(\phi_1,\phi_2)} \leftarrow 0) \vee (\delta'(\phi_1, \sigma, v_\Psi, \varepsilon_{\Psi^{Dup}}, f_\Upsilon) \wedge \mathit{Until}(\phi_1, \phi_2)/ \varepsilon_\Upsilon / f_{\mathit{Until}(\phi_1,\phi_2)} \leftarrow 1)$.

Informally speaking, the new transition function inhibits duplicates of the LASA every time a duplicate for that instance of the operator is already running. The automaton defined according to this semantics allows for a bounded number of duplicates, and can be proved (see below) to be equivalent to the one defined according to the standard semantics, i.e., they recognize the same language.

The optimizations included in the definition of Optimized Semantics 1 are valid under the assumption that the run-time verification framework is running in *standard* mode. A further optimization can be performed on the encoding of *Within* formulae, if the run-time verification framework operates in *critical*

[6] Due to lack of space, in the following we will only describe the elements of the definition that change in the proposed semantics. All the other cases remain as in the definition of the standard semantics, with Ψ_B^{Dup} and Υ_B^{As} empty, and the occurrences of δ changed into δ'.

Table 1. Comparison of the number of duplications of an alternating automaton, for the standard operational semantics and the proposed optimized semantics

Operator	# of duplications[a]		
	Standard Semantics	Optimized Semantics 1	Optimized Semantics 2
$\mathit{Until}(\phi_1, \phi_2)$	potentially infinite	1	1
$\mathit{Within}(\phi, K)$	N_K	N_K	1

[a] N_K represents the number of states in the sequence of process states that may occur in an interval of K time instances.

mode and thus the discovery of a violation of a specification leads to a complete halt in the verification activities. The following definition details these changes.

Optimized Semantics 2 (*critical* mode). Let ϕ be an ALBERT formula, and let $B = \{f_1, \ldots, f_n\}$ the set containing a boolean variable for each element in $\mathit{Sf}(\phi)$. The transition function δ'' of the LASA for ϕ is redefined[7] as follows, where $f_{\Psi^{Dup}} \in \Psi_B^{Dup}$ is a generic constraint on a variable $f_i \in B$:

- $\delta''(\mathit{Within}(\phi, K), \sigma, v_{\mathit{Within}(\phi,K)} = J, f_{\Psi^{Dup}}, f_{\Upsilon}) =$ $\mathit{false}/v_{\mathit{Within}(\phi,K)} \leftarrow 0/f_{\mathit{Within}(\phi,\ K)} \leftarrow 0$ if $J > K$;
- $\delta''(\mathit{Within}(\phi, K), \sigma, v_{\mathit{Within}(\phi,K)} = J, f_{\mathit{Within}(\phi,K)} = 0, f_{\Upsilon}) =$ $\delta''(\phi, \sigma, v_{\mathit{Within}(\phi,K)} = J, \varepsilon_{\Psi^{Dup}}, f_{\mathit{Within}(\phi,K)} \leftarrow 0) \vee \mathit{Within}(\phi, K)/v_{\mathit{Within}(\phi,K)} \leftarrow$ $(J + t(\sigma))/f_{\mathit{Within}(\phi,\ K)} \leftarrow 1$ if $J \leq K$.

Theorem 1. *Given an ALBERT formula ϕ, the ASA for ϕ obtained according to the definition of the standard semantics is always equivalent to the LASA defined according to Operational Semantics 1, and is equivalent to the LASA defined according to Optimized Semantics 2 only in* critical *mode.*

Proof. See [6].

Table 1 summarizes the gain, in terms of number of duplications, we can achieve for the *Until* and the *Within* temporal operators (and the ones derivable from them), by encoding ALBERT formulae in terms of LASA, as defined according to the two proposed optimized semantics.

5 Experimental Evaluation

We implemented the proposed encoding of ALBERT formulae within *Dynamo*, our run-time verification framework, by extending the existing component that

[7] Due to lack of space, in the following we will only describe the elements of the definition that change in the new semantics. All the other cases remain as in Optimized Semantics 1.

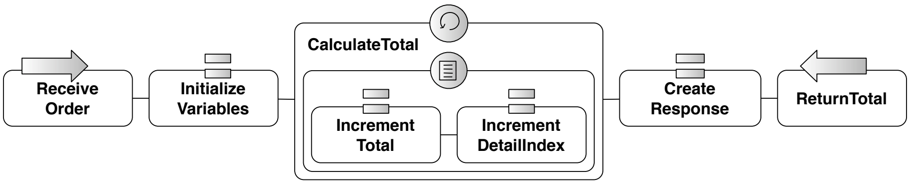


Fig. 3. The *While* sample BPEL process

is in charge of evaluating ALBERT formulae; more details about the architecture of the framework can be found in [2].

Our experiments were performed on a computer running Mac OS X 10.5.6 with a 2.16 GHz Intel Core 2 Duo processor and 2 GiB of memory. The current version of *Dynamo* is based on ActiveBPEL Community Edition 4.1; it was deployed on Apache Tomcat 5.5.27. Profiling data have been acquired by means of the profiler integrated in the NetBeans IDE.

For sake of simplicity and repeatability we chose to test our system with the *While* sample process, bundled with the ActiveBPEL distribution and shown in Fig. 3. This process uses an index variable to iterate over a list of order items and calculate the total cost of the order. The iteration is realized by means of *while* activity, and thus the name of the example. We made two simple modifications to the process: 1) Extending the number of iterations to 100, to increase the amount of time taken by the execution of the *while* activity. This can be seen as a simple way to simulate, using a simple toy example, the long asynchronous interactions that typically occur between a process and its partner services in the real-world. 2) Inserting copies of the index variable, to allow for writing several formulae of the same type, as explained below.

We chose to consider only properties containing *Until* formulae, to represent the worst-case execution scenario, where the number of duplications of the corresponding alternating automaton (and thus the number of threads required to evaluate the properties) is potentially infinite. The properties have the form

$$\bigwedge_{i=1}^{n} \mathit{Until}(\texttt{\$detailIndex}_i \geq 1, \mathit{onEvent}(\texttt{start\_ReturnTotal}))$$

where n, ranging from 1 to 10, represents the number of *Until* formulae to be evaluated at each run of the process, and the variables of the form $\texttt{\$detailIndex}_i$ are used by the process to implement the iteration. They are set to 1 when the process starts and are incremented once per iteration, meaning that the first sub-formula of the *Until* operator of the formula is always true. The second sub-formula, on the other hand, becomes true when the process executes the `ReturnTotal` *reply* activity.

The process has been executed on two different implementations: a non-optimized one based on the standard semantics defined in Sect. 3.2, and an optimized one, based on the definition of Optimized Semantics 2. On each im-

Table 2. Number of thread activations and time for the evaluation of the sample formulae

size of the property	non-optimized		optimized	
	# threads	time (ms)	# threads	time (ms)
0	49	155	48	160
1	324	8208	132	458
2	648	19061	217	711
3	1064	31628	358	748
4	1640	48550	371	952
5	N/A	N/A	579	1044
6	N/A	N/A	670	1358
7	N/A	N/A	794	1940
8	N/A	N/A	876	1879
9	N/A	N/A	1007	2254
10	N/A	N/A	1070	2180

plementation, we executed 11 experiments, one for each different size of the property (i.e., number of *Until* operators) plus one corresponding to the base case, with no properties to verify. For each experiment, the application container was restarted. Due to the complexity of the middleware infrastructure, measurements are not exactly reproducible; this is a well-known phenomenon, for example in Java-based environments, where measurement variances due to application-inherent non-determinism are often amplified by differences in thread scheduling, dynamic just-in-time compilation, or garbage collection [7]. In order to compensate for the measurement variances, we repeated each experiment 10 times (under the same settings) and reported the geometric mean of the 10 trials.

Table 2 shows the number of thread activations and the time required for the evaluation of the properties, each one with a different, increasing number of *Until* operators. In the non-optimized implementation, the number of threads continuously grows until the system runs out of memory and experiences failures (reported as Java exceptions) in the attempt to create new threads, thus making the number of threads and the evaluation time become irrelevant (and thus marked as N/A in the table). On the other hand, the optimized version behaves much better and terminates as expected. Notice that the optimized version still has some spurious threads that make the total number of threads greater than the value estimated by the theoretical model. These threads derive from implementation constraints that do not interfere with the proposed optimizations and that we plan to address in future versions of our prototype.

These results prove that the optimized implementation, which is based on the optimized semantics, is better than the non-optimized implementation, which is based on the standard semantics, since it allowed us to check for properties that we could not have checked using the other version.

6 Related Work

The approach most similar to ours is the one described in [8], which deals with the on-line monitoring of Service Level Agreements. The main difference lies in the kind of observable properties that are supported by the two frameworks, which impacts both on the underlying theoretical model and on the corresponding implementation. [8] supports only the specification of latency, reliability and throughput requirements, which are a subset of the properties that can be expressed with ALBERT. A consequence of this limitation is that they can lay the approach on the top of a simpler theoretical model (timed automata).

[9] proposes a complementary approach to the previous one, as it focuses on the run-time monitoring of safety and liveness properties of Web service interactions, and does not consider timeliness constraints. Properties are specified by means of UML 2.0 Sequence Diagrams (SD) that are then translated into non-deterministic finite automata, whose size is polynomial in the number of events and the number of processes described by an SD. This is comparable to the spatial complexity of our approach, as the size of an LASA is polynomial in the number of temporal operators of the corresponding formula.

In [2] we provided a detailed comparison of various approaches for the run-time verification of Web service compositions. None of them describes a formal model to reason about the efficiency of the proposed approach.

To the best of our knowledge, alternating automata (and their variations) have not been used before for the run-time verification of service interactions, i.e., in the context of multi-parties, distributed applications. However, they have been proposed for the run-time verification of stand-alone applications. For example, [10] uses Metric Temporal Logic (MTL), which shares many constructs with ALBERT, and represents a formula in terms of an evolving computation tree equivalent to the original alternating automaton corresponding to the formula. The proposed optimization, which identifies and eliminates redundant sub-structures of a computation tree, is somehow equivalent to our proposal of reducing the duplications of an automaton. [11] presents three algorithms, based on alternating automata, to check both Past and Future Time Linear Temporal Logic (LTL). The only algorithm suitable to work on-line, however, has an exponential space complexity in the size of the input formula.

All the approaches mentioned above work with infinite program traces. However, in some cases the execution traces may be finite and therefore special-purpose algorithm can be used. [12] proposes an approach for automata-based run-time verification, where the standard algorithm to convert (Future Time) LTL formulae to Büchi automata is modified to generate finite-state automata that check finite program traces. [13] uses non-deterministic Büchi automata to verify formulae written in TLTL, the timed version of LTL: in this case, the size of the automaton is exponential in the length of the corresponding formula as well as its largest constant.

7 Conclusions

The work presented in this paper demonstrates that ALBERT formulae can be represented by concise alternating automata with a bounded number of duplicates. This is achieved by introducing an extension of alternating automata and by providing the definition of an operational semantics of ALBERT formulae in terms of this model.

Presented results not only have mere theoretical consequences, but also are the basis for a concrete and efficient implementation of our run-time verification framework. Our future work will focus on the refinement of the current prototype implementation and on a thorough quantitative analysis of the run-time verification framework.

Acknowledgments. The authors wish to thank Carlo Ghezzi for his insightful comments on earlier versions of the paper. Part of this work has been supported by the Swiss NSF project "CLAVOS" and by the ERC grant 227977 "SMScom".

References

1. Andrews, T., et al.: Business Process Execution Language for Web Services, Version 1.1. BPEL4WS specification (2003)
2. Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., Spoletini, P.: Validation of web service compositions. IET Softw. **1**(6) (2007) 219–232
3. Bianculli, D., Ghezzi, C., Spoletini, P., Baresi, L., Guinea, S.: A guided tour through SAVVY-WS: a methodology for specifying and validating web service compositions. In: Advances in Software Engineering. Volume 5316 of LNCS. Springer (2008) 130–161
4. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. ACM **28**(1) (1981) 114–133
5. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Logics for concurrency: structure versus automata. Volume 1043 of LNCS., Springer (1996) 238–266
6. Baresi, L., Bianculli, D., Guinea, S., Spoletini, P.: Keep it small, keep it real: Efficient run-time verification of web service compositions. Technical Report 2009.9, Politecnico di Milano - Dipartimento di Elettronica e Informazione (2009)
7. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous Java performance evaluation. In: Proceedings of OOPSLA'07, ACM (2007) 57–76
8. Raimondi, F., Skene, J., , Emmerich, W.: Efficient online monitoring of web-service SLAs. In: Proc. of SIGSOFT '08/FSE-16, ACM (2008) 170–180
9. Simmonds, J., Chechik, M., Nejati, S.: Property patterns for runtime monitoring of web service conversations. In: Proc. of RV'08. (2008)
10. Drusinsky, D.: On-line monitoring of metric temporal logic with time-series constraints using alternating finite automata. JUCS **12**(5) (2006) 482–498
11. Finkbeiner, B., Sipma, H.B.: Checking finite traces using alternating automata. Electr. Notes Theor. Comput. Sci. **55**(2) (2001) 44–60
12. Giannakopoulou, D., Havelund, K.: Runtime analysis of linear temporal logic specifications. In: Proc. of ASE 2001, IEEE Computer Society (2001) 412–416
13. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Proc. of FSTTCS'06. Volume 4337 of LNCS., Springer (2006) 260–272