Enacting Declarative Languages using LTL: Avoiding Errors and Improving Performance

Maja Pešić[1], Dragan Bošnački[2], and Wil M.P. van der Aalst[1]

[1] Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB, The Netherlands
{m.pesic,w.m.p.v.d.aalst}@tue.nl
[2] Department of Biomedical Engineering,
Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB, The Netherlands
dragan@win.tue.nl

Abstract. In our earlier work we have proposed using the declarative language DecSerFlow for modeling, analysis and enactment of processes in autonomous web services. DecSerFlow uses constraints specified with Linear Temporal Logic (LTL) to implicitly define possible executions of a model: any execution that satisfies all constraints is possible. Hence, a finite representation of all possible executions is retrieved as an automaton generated from LTL-based constraints. Standard model checking algorithms for creating Büchi automata from LTL formulas are not applicable because of the requirements posed by the proper execution of DecSerFlow (and LTL-based process engines). On the one hand, LTL handles *infinite words* where each element of the word can refer to *zero or more propositions*. On the other hand, each execution of a DecSerFlow model is a *finite* sequence of *single events*. In this paper we adopt an existing approach to *finite-word* semantics of LTL and propose the modifications of LTL and automata generation algorithm needed to handle occurrences of *single events*. Besides eliminating errors caused by the 'multiple properties - single events' mismatch, the proposed adjustments also improve the performance of the automata generation algorithms dramatically.

1 Introduction

In our previous work [1] we have proposed using DecSerFlow for the specification, verification, monitoring and orchestration (i.e., execution) of web services in the context of business processes. DecSerFlow uses constraints to implicitly specify in which order tasks can be executed. Constraints are rules specified on two levels. First, there is a graphical representation which is similar to other graphical process modeling approaches: tasks are represented as rectangles and constraints as special lines between tasks. Second, Linear Temporal Logic (LTL) [5] is used for the formal specification of the semantics of these constraints. The graphical representation of a DecSerFlow model improves the readability of models, while

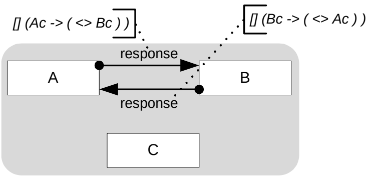


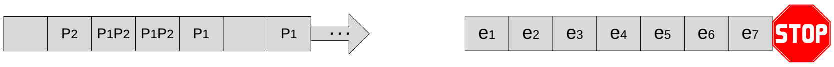
Fig. 1. A DecSerFlow model

(often complex and unreadable) LTL expressions enable verification, monitoring and deadlock-free execution.

Figure 1 shows an illustrative example of a DecSerFlow model. This model contains tasks A, B and C, and two *response* constraints. The first constraint specifies that each successfully completed execution of task A must eventually be followed by at least one successfully completed execution of task B, which is formally defined with the LTL formula $\Box(A_c \Rightarrow \Diamond B_c)$. The second constraint specifies that each successfully completed execution of task B must eventually be followed by at least one successfully completed execution of task A, which is formally defined by the LTL formula $\Box(B_c \Rightarrow \Diamond A_c)$. Note that DecSerFlow uses the notion of events related to execution of a task: scheduling, starting, completing, canceling, delegating, etc. For example, starting, canceling and completing the execution of some task T is denoted by T_s, T_x and T_c, respectively. This increases the expressive power of the language.

The DecSerFlow language is supported by the DECLARE system, which enables modeling, verification and enactment of LTL-based models, e.g., DecSerFlow models. Note that this system cannot directly be used to enact web services. Instead it can be used for manual execution of declarative models, where the user manually executes each step in the process. DECLARE is an open source system and it can be downloaded from http://declare.sf.net.

Using LTL for formalization of constraints makes DecSerFlow a truly declarative process modeling language, which increases flexibility [1] needed for autonomous web services. On the one hand, procedural process models lack flexibility because they explicitly specify all possible executions (i.e., orderings of tasks). On the other hand, DecSerFlow models are more flexible because all possible executions are specified implicitly, as all executions that satisfy all constraints in the model. A finite representation of all possible executions is obtained from LTL specifications of DecSerFlow constraints: by means of generating an automata for LTL formulas. Algorithms for generating automata that represent exactly all traces that satisfy an LTL formula have been developed in the field of model checking [5]. Note that, in DecSerFlow, we do not use automata generated from LTL specifications of constraints for model checking in the sense of checking if the model satisfies certain properties. Instead, we use the generated automata for detecting errors caused by conflicting constraints, execution monitoring and ensuring deadlock-free execution of models of web services [11].



(a) a standard LTL trace

(b) a DecSerFlow execution trace

Fig. 2. LTL for DecSerFlow

Typically, LTL and model checking techniques are used to verify properties of a given model. DecSerFlow uses LTL to define possible executions, as sequences of executed events. For example, one possible execution of the model shown in Figure 1 is executing (i.e., starting and completing) task C three times, which is specified as $\sigma = C_s, C_c, C_s, C_c, C_s, C_c$. There are *two important differences* between the 'standard' LTL and the LTL applied to DecSerFlow models, as is illustrated in Figure 2.

The first difference between standard LTL and DecSerFlow LTL is the length of execution traces (i.e., words). On the one hand, standard LTL considers infinite traces, as shown in Figure 2(a). On the other hand, Figure 2(b) shows that the execution of process instances (e.g. a customer order or service request) eventually terminates. Hence, the infinite semantics of standard LTL [5] cannot be applied to DecSerFlow models. In order to apply LTL to finite traces, we adopt a simple and efficient approach originally proposed by Giannakopoulou et al. [7].

The second difference between standard LTL and the DecSerFlow LTL is the semantics of elements in a trace. Standard LTL assumes that one element of the trace can refer to more than one *proposition*. For example, it is possible to monitor two properties: (P_1) the motor temperature is higher than 80 degrees and (P_2) the speed of the turbine is higher than 150 km/h. As Figure 2(a) shows, each element of the trace could then refer to: (1) none of the two properties, i.e., neither P_1 nor P_2 hold, (2) only property P_1 holds, (3) only property P_2 holds, or (4) properties P_1 and P_2 both hold. In the case of execution traces of DecSerFlow models one proposition refers to one event, e.g., starting task C (C_s). Therefore, we may safely assume that only one proposition holds at one moment, i.e., each of the elements of the trace refers to exactly one event, as shown in Figure 2(b).

Due to these differences, using standard LTL and automata generation algorithm may cause errors when it comes to the verification, monitoring and deadlock-free execution of DecSerFlow models. In this paper we show how the *semantics* of standard LTL and the automata generation can be adjusted for declarative languages like DecSerFlow. Besides elimination of errors, the proposed adjustments improve the performance of the algorithm, both with respect to the size of the automata and the processing time. We will use the model shown in Figure 1 as a running example.

The remainder of this paper is organized as follows. We start by describing how LTL and automata generated from LTL are used in DecSerFlow. In Section 3 we sketch how the finite trace semantics can be applied to LTL and generated automata, by using the approach described in [7]. Section 4 describes how the LTL semantics and the algorithm for generating automata must be changed in order to be applicable to sequences of *single events*. The results of experiments testing the performance of proposed changes are presented in Section 5. Finally, related work is discussed in Section 6 and Section 7 concludes the paper.

2 LTL and Automata for Finite Traces

DecSerFlow uses LTL for finite traces [7] to formally specify constraints. Just like in standard LTL, a well-formed LTL formula can use standard logical operators (!, $\wedge$ and $\vee$) and several additional temporal operators: $\bigcirc$ (next), U (until), W (weak until), V (release), $\Box$ (always) and $\Diamond$ (eventually). The finite trace semantics is reflected in the way propositions and the until (U) operator are defined: proposition is a finite sequence and the index i in the until (U) has the upper bound n:

Definition 1 (LTL for finite traces [7]). *Given a finite set of atomic propositions P, every $p \in P$ is a well-formed LTL formula. If Φ and Ψ are well-formed LTL formulas, then* `true`*,* `false`*, $!\Phi$, $\Phi \wedge \Psi$, $\Phi \vee \Psi$, $\Box\Phi$, $\Diamond\Phi$, $\bigcirc\Phi$, $\Phi U \Psi$, $\Phi V \Psi$ and $\Phi W \Psi$ are also well-formed LTL formulas. An interpretation of an LTL formula is a set of finite traces $\sigma = \sigma_1, \sigma_2, \ldots$ over 2^P (sets of propositions). We write $\sigma^{i\rightarrow}$ for the suffix of σ starting at position i, i.e., $\sigma^{i\rightarrow} = \sigma_i, \sigma_{i+1}, \ldots$. The semantics of LTL is defined as follows:*

- $\sigma \models p$ *iff* $p \in \sigma_1$, *for* $p \in P$
- $\sigma \models \Phi \vee \Psi$ *iff* $(\sigma \models \Phi) \vee (\sigma \models \Psi)$
- $\sigma \models !\Phi$ *iff* $\sigma \not\models \Phi$
- $\sigma \models \bigcirc\Phi$ *iff* $\sigma^{2\rightarrow} \models \Phi$
- $\sigma \models \Phi \wedge \Psi$ *iff* $(\sigma \models \Phi) \wedge (\sigma \models \Psi)$
- $\sigma \models \Phi U \Psi$ *iff* $(\exists_{1 \leq i \leq n} : (\sigma^{i\rightarrow} \models \Psi \wedge (\forall_{1 \leq j < i} : \sigma^{j\rightarrow} \models \Phi)))$

Also, abbreviations are used:

- $\Phi \Rightarrow \Psi$ *for* $!\Phi \vee \Psi$
- $\Diamond\Phi$ *for* `true`$U\Phi$
- $\Phi W \Psi$ *for* $(\Phi U \Psi) \vee (\Box\Phi)$
- `true` *for* $\Phi \vee !\Phi$
- $\Box\Phi$ *for* $!\Diamond!\Phi$
- $\Phi V \Psi$ *for* $!(!\Phi U !\Psi)$
- `false` *for* !`true`

DecSerFlow does not directly use LTL for constraint specification. Instead, constraints are created from constraint templates. Each template has a unique name and graphical representation and its semantics is formalized by an LTL formula. Although DecSerFlow has more than twenty constraint templates, new templates can be easily added to the language. Hence, any LTL formula can be used in DecSerFlow. For the detailed description of the full list of DecSerFlow templates we refer the reader to [11].

A widely exploited property of LTL is the fact that for every LTL formula a finite state automaton (cf. Definition 2) can be generated, such that the language (i.e., all accepted traces) of this automaton exactly represents all traces that satisfy the formula. In the field of model-checking, various algorithms are proposed for generating automata from LTL formulas. Some examples of these algorithms can be found in [5, 8, 3, 4, 7]. In the remainder of this paper we use the algorithm for generating finite automata for finite traces presented in [7]. We will refer to this algorithm as to the `BASIC` algorithm.

Definition 2 (Finite automaton (*FA*)). *Finite automaton FA is a five-tuple* $\langle A, S, T, S_0, S_F \rangle$ *such that A is the* alphabet*, S is the finite set of* states*,* $T \subseteq S \times A \times S$ *it the* transition relation*,* $S_0 \subseteq S$ *is the set of* initial states*, and* $S_F \subseteq S$ *is the set of* accepting states*. We say that:*

- *A run of FA is a sequence* $\delta = s_0 a_0 s_1 \ldots a_{n-1} s_n$ *such that* $(s_i, a_i, s_{i+1}) \in T$ *and* $s_0 \in S_0$*.*
- *A run* δ *is accepting if* $s_n \in S_F$*.*
- *A FA accepts a finite trace* $\sigma = \langle e_0, e_1, \ldots, e_m \rangle$ *if there exists an accepting run* $\delta = s_0 e_0 s_1 \ldots e_{m-1} s_m$ *of FA.*

All algorithms (including the `BASIC` algorithm) are based on expanding a graph node into a set of nodes, which will eventually become states of the automaton [5, 8, 3, 4, 7]. Each node has several fields [5]. Field *ID* is a unique identification for the node; Field *INCOMING* contains the set of nodes that lead to this node; Field *NEW* contains the set of formulas that must hold in the current node but have not yet been processed, i.e., this node must make these formulas true; Field *OLD* contains the set of formulas that have already been processed; Field *NEXT* contains the set of formulas that must hold at all immediate successors of this node. The `BASIC` algorithm consists of several basic steps:

1. The original LTL formula is rewritten to a normal form.
2. A graph of nodes is created by the recursive method *expand*, which is briefly sketched in Algorithm 2.1. Formulas from the field NEW are processed one by one (line 14), by breaking down the formulas to the level of propositions. Formulas aUb, aVb and $a \vee b$ are broken down by creating two new nodes (following special rules) and expanding them further (lines 24-27) and formula $a \wedge b$ by adding a and b to the field NEW and further expanding the current node (lines 29-31). While processing a literal f (lines 16-23), a conflict occurs and the current node is discarded if the field OLD of the current node already contains $!f$ (lines 17 and 18). When there is no conflict, f is added to the field OLD and the current node is further expanded (lines 20 and 21). Expanding a specific node finishes when all formulas have been processed, i.e., the NEW field is empty (lines 2-12). If an equal node is already in the GRAPH, then this (equal) node is updated with data for the current node (lines 3-6). Otherwise, the current node is added to the GRAPH, a new node is created and expanded with the current NODE in the field INCOMING and all formulas from the field NEXT of the current node in the field NEW of the created node(lines 7-11).
3. A finite automaton is created in the following manner: (1) created nodes become states in the automaton, (2) automaton edges are defined by the INCOMING field, (3) labels on edges are defined by literals stored in the field OLD, and (4) finite acceptance of states is imposed, ensuring that an accepting node does not have any unprocessed until (U) formulas in the NEXT field.

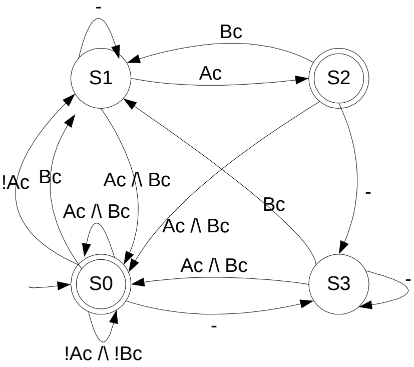
The procedure for creating a graph nodes in the BASIC algorithm is also part of standards algorithms [5, 8, 3, 4, 7]. The main difference is in the way acceptance of nodes (i.e., states) is determined in the last step. On the one hand, standard algorithms impose infinite acceptance ensuring that, whenever a node contains pUq, some successor node will contain q [5]. On the other hand, the BASIC algorithm imposes finite acceptance by ensuring that an accepting trace satisfies all required eventualities, i.e., the NEXT field does not contain any unprocessed until (U) formulas [7].

Algorithm 2.1 BASIC algorithm: expanding the graph of nodes

```
function expand(NODE, GRAPH)
if $NODE.NEW = \emptyset$ then {*finished processing node*}
  if $\exists A \in GRAPH : equal(NODE, A)$ then {*equivalent node already processed*}
    update_existing(NODE,A); {*just update the existing node*}
    return GRAPH;
  else
    $GRAPH \Leftarrow GRAPH \cup \{NODE\}$; {*add NODE to GRAPH*}
    $NEWNODE \Leftarrow createNewNode()$; {*create NEWNODE*}
    $NEWNODE.INCOMING \Leftarrow \{NODE\}$;
    $NEWNODE.NEW \Leftarrow NODE.NEXT$;
    return $expand(NEWNODE, GRAPH)$; {*expand NEWNODE*}
  end if
else
  $f \Leftarrow getFormula(NODE.NEW)$; {*get the next formula for processing*}
  $NODE.NEW \Leftarrow NODE.NEW \setminus \{f\}$;
  if $(f \in P \vee !f \in P) \vee (f = \texttt{true} \vee f = \texttt{false})$ then {*f is a literal*}
    if $(f = \texttt{false}) \vee (!f \in NODE.OLD)$ then {*a contradiction in NODE*}
      return GRAPH {*discard current NODE*}
    else
      $NODE.OLD \Leftarrow NODE.OLD \cup \{f\}$
      return expand(NODE,GRAPH)
    end if
  end if
  if $f = aUb$, $aVb$, or $a \vee b$ then
    $NODE1 \Leftarrow create1(f, NODE)$; {*depending the current operator in f*}
    $NODE2 \Leftarrow create2(f, NODE)$; {*depending the current operator in f*}
    return expand(NODE2,expand(NODE1, GRAPH));
  end if
  if $f = a \wedge b$ then
    $NODE.NEW \Leftarrow NODE.NEW \cup \{a, b\}$;
    return expand(NODE,GRAPH);
  end if
end if
end expand;
```

$$F = (\Box(A_c \Rightarrow \Diamond B_c)) \wedge (\Box(B_c \Rightarrow \Diamond A_c))$$

(a) model formula



(b) model automaton generated using the `BASIC` algorithm

Fig. 3. Retrieving a finite representation of all possible executions of the model shown in Figure 1

A finite representation of all possible executions of a DecSerFlow model (i.e., all executions that satisfy all constraints in the model) is obtained by generating the *model automaton* from the *model formula*. The model formula F for a model with n constraints with LTL formulas $f_1, f_2, \ldots, f_n$ is a conjunction of LTL formulas for all constraints from the model, i.e., $F = f_1 \wedge f_2 \wedge \ldots \wedge f_n$. For example, the *model formula* for the DecSerFlow model shown in Figure 1 is a conjunction of formulas for the two *response* constraints, as shown in Figure 3(a). Figure 3(b) shows the Büchi automaton generated by the `BASIC` algorithm for this model formula. Hence, the language (i.e., all accepted traces) of this automaton represents all possible executions of the DecSerFlow model from Figure 1. Automaton states are represented by circles such that a single border marks a non-accepting and a double border accepting state. Transitions are represented as directed labeled arcs between states. The arc without a source state marks the initial state.

Automata generated from DecSerFlow models can be used for multiple purposes [1, 11]. For example, the *model automaton* and automata generated for each constraint can be used to monitor the state of a model instance (we refer to one execution of a model as to one instance of the model) and constraints during execution. This is done by checking if the trace satisfies the model, i.e., if the automaton accepts the trace. Naturally, when processing the instance state the model automaton is used, and when processing states of constraints automata generated from LTL specifications are used. Note that the generated automata are non-deterministic, and we say that an automaton accepts a trace if the trace can be 'replayed' on the automaton in such a way that an accepting state is reached [1]. Given the current execution trace of an instance, the instance or constraint state is determined as follows:

- If the trace is accepted by the automaton, then the instance/constraint is *satisfied*.
- The instance/constraint is *temporarily violated* if the current trace is not accepted but it is a prefix of a trace accepted by the automaton. In other words, the instance/constraint is *temporarily violated* if the current trace can be 'replayed' on the automaton, but all possible replay scenarios lead to non-accepting state.
- If the trace is neither accepted by the automaton nor it is a prefix of an accepted trace, then the instance/constraint is *(permanently) violated*. In other words, the instance/constraint is *violated* if the current trace can not be 'replayed' on the automaton at all.

In addition to state monitoring, the model automaton can be used to ensure a deadlock-free execution and to verify service models. On the one hand, if the service execution would be driven by the model automaton, deadlocks would be eliminated. On the other hand, we have developed a verification procedures that can detect two types of errors based on model automata. First, a dead task is a model task that can never be executed because its completion is never allowed by transition labels. Second, a model has a conflict if the generated automaton is empty, i.e., it has no states and its language is empty. The DECLARE system used the above described procedures for model verification, monitoring states of instances and constraints and ensuring the deadlock-free execution [11].

3 Applying the Finite Traces Semantics

As explained in Section 1, an execution trace of a web service is a *finite* sequence, i.e., $\sigma = p_1, p_2, \ldots p_n$. Using automata generated for infinite traces can create problems if used for finite executions of DecSerFlow models.

Consider, for example, the situation when the DecSerFlow model shown in Figure 1 is executed by executing task A, i.e., the execution trace is

$$\sigma = A_s, A_c. \tag{1}$$

The model automaton shown in Figure 3(b) suggests that σ satisfies the model because it brings the model automaton to the accepting state S_2 by triggering transitions '$!A_c$' and 'A_c': $S_0 \xrightarrow[(!A_c)]{A_s} S_1 \xrightarrow[(A_c)]{A_c} S_2$.

If the finite semantics of the until (U) operator is assumed, trace σ given in (1) does not satisfy the model formula shown in Figure 3(a). This is because event A_c is not followed by event B_c by the end of the trace (note that this is required by constraint *response(A,B)*). Hence, trace σ does not satisfy the model shown in Figure 1. However, as explained in Section 2, the model automaton shown in Figure 3(b) suggests otherwise: trace σ satisfies the model because it brings the model automaton to the accepting state S_2.

To avoid this type of errors, the algorithm for automata generation must be adjusted to finite traces, as described in [7]. The infinite semantics of LTL is

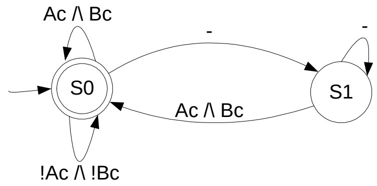


Fig. 4. The $\texttt{BASIC}^{FIN}$ model automaton for model shown in Figure 1

reflected in the fact that i does not have an upper bound in the definition of the until (U) operator: $\exists_{1\leq i} : \ldots$ (cf. Section 2). The manner in which the infinite acceptance is imposed in the standard algorithm is described in Section 2. As described in [7], finite semantics must be reflected in the upper bound n of i in the until operator: $\sigma \models \varphi U \psi$ if and only if $(\exists_{1\leq i\leq n} : (\sigma^i \models q \wedge (\forall_{1\leq j<i} : \sigma^j \models p))$. The main change in the algorithm is the way accepting conditions are imposed: a finite trace is accepting only if it satisfies all required eventualities (i.e., untils). Formulas that still need to be satisfied are stored in the field NEXT. Therefore, only if the NEXT field of a node does not contain any until (U) formulas, the automaton state generated from this node is accepting [7]. In the remainder of this paper we will use $\texttt{BASIC}^{FIN}$ to denote the algorithm presented in [8] and modified for finite traces as described in [7]. We will refer to the automaton generated by the $\texttt{BASIC}^{FIN}$ algorithm as to the $\texttt{BASIC}^{FIN}$ automaton. Figure 4 shows the $\texttt{BASIC}^{FIN}$ automaton generated for the model formula given in Figure 3(a). Indeed, this automaton suggests that trace σ does not satisfy the model from Figure 1, because this trace brings the automaton to the non-accepting state S_1: $S_0 \xrightarrow[(!A_c \wedge !B_c)]{A_s} S_0 \xrightarrow[(-)]{A_c} S_1$.

4 Applying the 'Single Event' Semantics

In standard LTL, traces are defined over 2^P, which means that σ is a sequence of sets of propositions ($\forall_{1\leq\ i} : \sigma_i \subseteq P$). Hence, each element σ_i of a trace is a *set of atomic propositions*. The fact that one element of a trace can refer to multiple properties in standard LTL is semantically expressed in the way the proposition is defined: $\sigma \models p$ if and only if $p \in \sigma_1$ (cf. Section 2). As explained in Section 1, execution trace of a web service is a sequence of *single* events/propositions: $\forall_{1\leq i\leq n} : \sigma_i \in P$. In order to adjust the semantics of LTL to traces where each element refers to exactly one event, i.e. proposition, we must check if the proposition *is* the first element of the trace: $\sigma \models p$ if and only if $p = \sigma_1$.

Automata that consider traces containing sets of propositions can create problems in DecSerFlow. For example, consider the $\texttt{BASIC}^{FIN}$ model automaton shown in Figure 4 and an instance of the model (cf. Figure 1 on page 2) with the execution trace σ given in (1). As explained in Section 3, this automaton suggests that the instance is not satisfied because trace σ brings the automaton to the non-accepting state S_1. Moreover, because an accepting state is reachable (i.e.,

Algorithm 4.1 Detecting a contradiction in the 'single events' algorithm

```
 1: function expand(NODE, GRAPH)
 2: ...
 3: if $(f \in P \vee !f \in P) \vee (f = \mathtt{true} \vee f = \mathtt{false})$ then
 4:    if $(\ f = \mathtt{false}) \vee (!f \in node.OLD) \vee (l \in P \wedge f \neq l \wedge l \in node.OLD)$ then
 5:       return GRAPH
 6:    else
 7:       $NODE.OLD \Leftarrow NODE.OLD \cup f$
 8:       return expand(NODE,GRAPH)
 9:    end if
10: end if
11: ...
12: end expand;
```

accepting state S_0 is reachable from S_1 via transition $A_c \wedge B_c$), this automaton suggests that the instance is only temporarily violated. However, because events are triggered *one by one*, transition $A_c \wedge B_c$ can never be taken. Hence, the state of this instance is actually *permanently violated* because an accepting state can no longer be reached.

In order to eliminate this error, we must adjust the algorithm to consider sequences of single events in the following way. The most important change is strengthening the contradiction test in Algorithm 2.1 (lines 16-18). As described in Section 2, literals that belong to the field *OLD* will become labels on the transitions in the generated automaton. Therefore, if the processed formula f is a literal, contradiction occurs and the current node is discarded if negation of f ($!f$) is already in the field *OLD* (lines 13 and 14).

Algorithm 4.1 shows how contradiction requirements must be strengthened in order to reflect the single event property. The additional requirement is: if we are processing a proposition f and another proposition $l \neq f$ is already in the field *OLD*, then this is also a contradiction (line 3). Further, the handling of contradictions and regular situations stays the same: the current node is discarded when a contradiction is detected. Otherwise, further expansion of the current node in the graph is continued.

In addition to strengthening the contradiction requirement, labels on transitions can be displayed in a more concise way. If a label contains one positive proposition and an arbitrary number of negative proposition (e.g., $A_c \wedge !B_c \wedge !C_c$), it can be replaced by a shorter label containing only the positive proposition (e.g., A_c). This is because the latter is implied by the former. Note that making labels shorter is not necessary from the semantical and correctness perspective, but it significantly improves the readability.

Figure 5 shows the $\mathtt{BASIC}_{SE}^{FIN}$ automata generated for the model formula given in Figure 3(a). This automaton correctly indicates that an instance with trace σ is *permanently violated* because σ is neither accepted by the automaton, nor it is a prefix of a trace accepted by this automaton. Moreover, tasks A and B can never be executed (i.e., these are dead tasks) because the language of this automaton does not accept traces that contain A_c or B_c. This is because, as soon as either A or B would be executed, no finite execution would be able to satisfy *both response* constraints from the model shown in Figure 1.

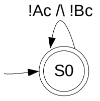


Fig. 5. The $\texttt{BASIC}^{FIN}_{SE}$ model automaton for model shown in Figure 1

4.1 Correctness Arguments

In this section we give a brief discussion of the correctness of the single event Algorithm 4.1 $\texttt{BASIC}^{FIN}_{SE}$. Algorithm 2.1, $\texttt{BASIC}^{FIN}$, can also be used to generate automata that accept only single-event traces satisfying a given model formula F. To this end we need to add a restrictive conjunct to F. Thus, to rule out all multiple-event traces, we run $\texttt{BASIC}^{FIN}$ on the formula $R \wedge F$, where $a_1, a_2, \ldots, a_n \in P$ are the events that appear in F, and $R = \Box \bigwedge_{i,j} !(a_i \wedge a_j)$ where $1 \leq i \leq n$, $1 \leq j \leq n$ and $i \neq j$.

For the correctness of the standard algorithm $\texttt{BASIC}^{FIN}$ we rely on [5, 8]. Hence, we can establish the correctness of the single-event algorithm $\texttt{BASIC}^{FIN}_{SE}$ by showing the following: for each run $\mathcal{R}_2$ of the algorithm $\texttt{BASIC}^{FIN}_{SE}$ applied to a given model formula F, there exists a run $\mathcal{R}_1$ of algorithm $\texttt{BASIC}^{FIN}$ applied to the formula $R \wedge F$, such that the automata produced by $\mathcal{R}_2$ and $\mathcal{R}_1$ are equivalent, i.e., they accept the same language. Moreover, to each state of the automaton generated by $\texttt{BASIC}^{FIN}_{SE}$ there corresponds an equivalence class of states in the automaton generated by $\texttt{BASIC}^{FIN}$.

To show this, we construct a run $\mathcal{R}_1$ of algorithm $\texttt{BASIC}^{FIN}$ as we trace the run $\mathcal{R}_2$ of algorithm $\texttt{BASIC}^{FIN}_{SE}$. In $\texttt{BASIC}^{FIN}_{SE}$ run $\mathcal{R}_2$ is applied to F', which is the normal form of F. Run $\mathcal{R}_1$ simulates $\mathcal{R}_2$ in a "stuttering" manner, i.e., multiple steps of $\mathcal{R}_1$ can correspond to a single step of $\mathcal{R}_2$. Algorithm $\texttt{BASIC}^{FIN}$ is actually applied on the normal form of $R \wedge F$, where F is rewritten to its normal form F' an R is rewritten into $R' = \texttt{false}\ V \bigwedge_{i,j}(!a_i \vee !a_j)$ such that $1 \leq i \leq n$, $1 \leq j \leq n$ and $i \neq j$. In $\mathcal{R}_1$ we process R' before F'. It is straightforward to check that a sequence of node transformations as a result of the processing of R' in $\mathcal{R}_1$ leads to insertion of $n - 1$ literals of the form $!a_i$ in the field OLD of each generated node [5]. This ensures that at most one positive proposition a_k can be added to the OLD field of each node, such that $!a_k \notin OLD$. Adding the second positive proposition a_l $(l \neq k)$ will automatically invoke a contradiction in the original sense because the OLD filed already contains the negation $!a_l$.

At each point one can prove that the following invariant holds for the parallel execution of $\mathcal{R}_1$ and $\mathcal{R}_2$. Let run $\mathcal{R}_2$ discard its current node n_2, because field OLD contains proposition l (lines 4-5 in $\mathtt{BASIC}_{SE}^{FIN}$) different from the currently processed formula/proposition f. Then field OLD of node n_1, currently considered by run $\mathcal{R}_1$, contains $!f$. As a consequence, $\mathcal{R}_1$ also discards the node (because of contradiction) and in this way rules out a possible multiple event transition in the automaton.

Besides that, one can show that for each action by $\mathcal{R}_1$ that adds a new node n_1 to the set of nodes (lines 7-11 in $\mathtt{BASIC}^{FIN}$), there exists an action of $\mathcal{R}_2$ that adds a corresponding node n_2. Both n_1 and n_2 contain in OLD only one literal without negation and this literal is the same in both nodes. Nodes n_1 and n_2 will be transformed into equivalent states in the resulting automata. Vice versa, each node added by $\mathcal{R}_2$ corresponds to an equivalent node added by $\mathcal{R}_1$.

5 Experiments

We performed experiments to measure the effects of the 'single events' adjustments on the performance of the $\mathtt{BASIC}_{SE}^{FIN}$ algorithm compared to the $\mathtt{BASIC}^{FIN}$ algorithm. We run $\mathtt{BASIC}^{FIN}$ with LTL formulas which were conjoined with a corresponding restrictive conjunct R, as introduced in Section 4.1 above, i.e., $R = \Box \bigwedge_{i,j} !(a_i \wedge a_j)$ where $1 \leq i \leq n$, $1 \leq j \leq n$ and $i \neq j$. Using such modified formulas can be seen as a high-level implementation of the single event restriction, i.e., without modification of the algorithm $\mathtt{BASIC}^{FIN}$ and its implementation.

We used the testing method based on randomly generated LTL formulas presented in [3, 8]. Each test set consists of F randomly generated LTL formulas of length L with N propositional variables. Temporal operators U and V are generated with the probability P. A formula of length L is generated in the following way:

$\boldsymbol{L} = 1$ Randomly generate a propositional variable using a uniform distribution.

$\boldsymbol{L} = 2$ Randomly generate a unary operator from the set $\{!, \bigcirc\}$. Apply the generated unary operator to a random formula of $L = 1$.

$\boldsymbol{L} > 2$ Randomly generate an operator from the set $\{!, \bigcirc, \vee, \wedge, U, V\}$. The probability to generate either U or V is $\frac{P}{2}$ and $\frac{1-P}{4}$ to generate the other operators. If the chosen operator is unary, it is applied to a random formula of $L = 1$. If the chosen operator is binary, it is applied to two random formulas: one of length S and the other of length $L - S - 1$. S is generated randomly using a uniform distribution between 1 and $L - 2$ inclusive.

We performed tests on ten sets containing $F = 100$ randomly generated formulas of varying lengths with 5 propositional variables ($N = 5$) and the probability of $\frac{1}{2}$ to select operators U and V. In each test set, formulas had different lengths: $L \in \{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$. As mentioned above, for each formula f, we apply directly $\mathtt{BASIC}_{SE}^{FIN}$ to f, whilst $\mathtt{BASIC}^{FIN}$ is applied to the f extended with its corresponding restricting conjunct to ensure the single event semantics.

When evaluating algorithms for generating automata from LTL formulas, it is a common practice to consider the size of the automaton (i.e., number of states and transitions) and the total time needed to generate the automaton [3, 8]. In addition to measuring the automata size and processing time, we have measured the number of times the procedure for *expanding* a node was invoked. This is because strengthening the contradiction requirements with the single event property causes the algorithm to abandon more expansion paths in the graph. Hence, the *expand* procedure is typically invoked fewer times.

Figure 6 presents the ratio between the corresponding results of the $\mathtt{BASIC}_{SE}^{FIN}$ algorithm and the $\mathtt{BASIC}^{FIN}$ algorithm. The results show that the low level implementation of the single event semantics, featured in $\mathtt{BASIC}_{SE}^{FIN}$, significantly improves the performance by reducing the processing time, automata size, and number of invocations of the *expand* procedure. For example, only 10% of the original time is needed to process a formula of length 35 and 45. A lower number of invocations of procedure *expand* results in shorter processing times. The typical automata size is also reduced because transitions referring to more than one property are eliminated.

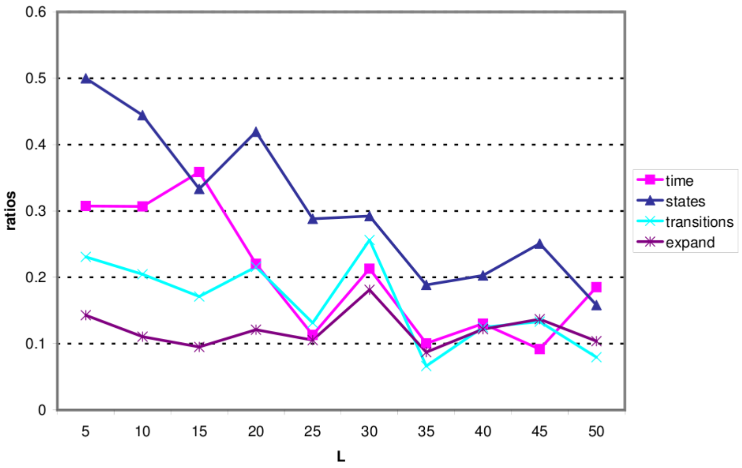


Fig. 6. The ratio of number of states, number of transitions, number of node expansion, and generation time between the $\mathtt{BASIC}_{SE}^{FIN}$ and $\mathtt{BASIC}^{FIN}$ algorithms for $F = 100$ randomly generated formulas of various lengths L, with $N = 5$ and $P = \frac{1}{2}$

Figure 6 shows that effects become more significant as the length of the formula increases up to the values $L = 25$. With further increase of the formula length, it seems that the effect begins to stabilize.

Each improvement of the performance with the $\texttt{BASIC}^{FIN}_{SE}$ algorithm is important for enactment of declarative models. This is because the model formula of a DecSerFlow model is generated as a conjunction of all constraints and, hence, it can be much longer than typical formulas used in model checking. If $L(f)$ denotes length of LTL formula f, then the length of model formula $F = c_1 \wedge c_2 \wedge \ldots \wedge c_n$ of a DecSerFlow model with n constraints specified with LTL formulas $c_1, c_2, \ldots, c_n$ is $L(F) = n-1+\sum_{i=1}^{n} L(c_i)$. For example, the length of model formula F (cf. Figure 3(a)) is $L(F) = 2 - 1 + \sum_{i=1}^{2} 5 = 11$, because the length of each *response* constraint in the model shown in Figure 1 is $L(c) = 5$.

6 Related Work

In our previous work, we have proposed using DecSerFlow for declarative specification of web service processes [1]. The $\mathcal{S}$CIFF language is another declarative language [2], which is based on abductive logic programming. While DecSerFlow and $\mathcal{S}$CIFF are similar with respect to their declarative approach to process modeling, both languages have some specific advantages. $\mathcal{S}$CIFF has more expressive power and is more efficient while checking constraints on executed traces (i.e., a posteriori) [11]. However, $\mathcal{S}$CIFF cannot ensure a deadlock-free execution at run-time. Moreover, the adjustments proposed in this paper enable detection of more sophisticated model errors and improves efficiency.

DecSerFlow uses LTL for formal specification of constraints and automata generated from LTL formulas for retrieving the final representation of all possible model executions as all executions that satisfy model constraints. LTL is extensively used in the field of model checking and algorithms for automata generation from LTL formulas based on [5] are proposed in this field. Moreover, improving the performance of these algorithms is an important topic in the field [3, 8, 4]. While these approaches improve the performance of the original algorithm, which works with the standard LTL, changes proposed in this paper improve the performance by limiting the LTL to sequences of single events.

The problem of applying finite traces semantics to standard LTL has been addressed by other researchers. Approach presented in [10] considers only safety properties and generating finite-trace automata for monitoring running programs. Adjustments of the algorithm for automata generation to finite traces is given in [7]. We use this approach because it does not limit the expressiveness of DecSerFlow [7].

Standard LTL considers properties, rather than events, which means that zero or more properties can hold at any point of time. Methods for model checking for event-based systems use several approaches to LTL for sequences of single events. In [12], an approach is proposed for specifying events indirectly in terms of edges. Edges do not relate directly to occurrence of event, but capture changes of truth/false values of atomic propositions. The Tracta model-checking approach

[6] for analysis of concurrent systems uses the special kind of LTL for sequences of single actions: Action Linear Temporal Logic (ALTL). However, this approach uses the Büchi automata following the standard automata-theoretic approach to verification and focuses on solving issues related to hierarchical systems using the Compositional Reachability Analysis (CRA) [6]. ALTL is extended for *fluent model checking* in [9]. Here, instead of using each event occurrence, time intervals between action initiation and termination are considered. A special model checking procedure is proposed for fluent actions. This procedure uses Büchi automata, but avoids the need for using the synchronous product operation [9].

7 Conclusions

Using standard LTL and Büchi automata for enacting DecSerFlow models can cause errors and inefficiencies. This is because of two important differences between the standard model checking problem and the execution processes based on a declarative language like DecSerFlow. This paper describes how the standard LTL and algorithm for automata generation can be adapted in order to fit two special properties of DecSerFlow. Both the adjustments for finite traces described by Giannakopoulou et al. in [7], and the adjustments for sequences for single events described in Section 4 must be used in order to avoid errors. Because automata are generated for the DecSerFlow model formula (which is a conjunction of formulas for all constraints), the performance of the algorithm becomes a potential bottle neck. For the special class of problems where properties hold one at a time, results of our experiments show that the proposed adjustments also significantly decrease the processing time and size of automata. Processing times are reduced for more than one order of magnitude. Since the enactment of declarative languages like DecSerFlowrequires the repeated execution of this procedure (each time some activity is started or completed), this is highly relevant.

References

1. W.M.P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In M. Bravetti, M. Nunez, and G. Zavattaro, editors, *International Conference on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 1–23, Vienna, Austria, 2006. Springer-Verlag.
2. M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, S. Storari, and P. Torroni. Computational Logic for Run-Time Verification of Web Services Choreographies: exploiting the SOCS-SI tool . In *Proceedings of the 3rd International Workshop on Web Services and Formal Methods (WS-FM06)*, volume 4184 of *Lecture Notes in Computer Science*, pages 58–72. Springer Verlag, Berlin, 2006.
3. M. Daniele, F. Giunchiglia, and M.Y. Vardi. Improved Automata Generation for Linear Temporal Logic. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99)*, pages 249–260, London, UK, 1999. Springer-Verlag.

4. P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, pages 53–65, London, UK, 2001. Springer-Verlag.
5. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd.
6. D. Giannakopoulou. *Model Checking for Concurrent Software Architectures* . PhD Thesis, University of London, London, The United Kingdom , 1999.
7. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, pages 412–416, Washington, DC, USA, 2001. IEEE Computer Society.
8. D. Giannakopoulou and F. Lerda. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems (FORTE '02)*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326, London, UK, 2002. Springer-Verlag.
9. D. Giannakopoulou and J. Magee. Fluent Model checking for event-based systems . In *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of Software Engineering (ESEC/FSE-11)*, pages 257–266. ACM, 2003.
10. T. Latvala. Efficient Model Checking of Safety Properties. In *Proceedings of the 10th SPIN Workshop on Model Checking of Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 74–88. Springer Verlag, Berlin, 2003.
11. M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative specification and verification of service choreographies. *ACM Trans. Web*, 4(1):1–62, 2010.
12. D.O. Pbreveaun and M. Chechik. Events in Linear-Time Properties. In *Proceedings of the 4th IEEE International Symposium on Requirements Engineering (RE 99)*, Lecture Notes in Computer Science, pages 123–132. IEEE Computer Society, 1999.