A Solution to the State Space Explosion Problem in Declarative Business Process Modeling

Renata M. de Carvalho, Natalia C. Silva, Cesar A. L. Oliveira, Ricardo M. F. Lima
Center for Informatics, Federal University of Pernambuco, Brazil
{rwm, ncs, calo, rmfl}@cin.ufpe.br

Abstract

Declarative business process models focus on modeling what must be done but do not determine how. The existing engine for controlling the execution of declarative processes uses automata-based model checking. Unfortunately, the well-known state space explosion problem limits the ability to explore large processes through automata-based approaches. In this work, we propose a novel mechanism to control the execution of declarative business processes. Our approach has the advantage of not requiring the computation of all reachable states. This allows for the modeling and execution of larger business processes when compared to the automata-based approach.

1 Introduction

Declarative business processes surged from the necessity for supporting process execution in complex or changing environments [4]. The declarative approach describes business processes by means of *business rules* that state what one can, can not, or should do to produce the desired output. It informs *what* has to be done, but not *how*.

The first work to propose the declarative paradigm of process modeling was published by Pesic [4]. Her work also proposes the *DECLARE* system, which is a tool capable of modeling and interpreting declarative process models [4]. *DECLARE* employs Linear Temporal Logic (LTL) to formally model and verify business rules. In order to control the execution of a business process, *DECLARE* converts the LTL formula into a finite non-deterministic automaton (FNDA). This automaton contains all possible execution paths for the business process, according to its rules.

Unfortunately, the number of states in an automaton increases exponentially with the size of the LTL formula [2]. As the number of business rules increases, the corresponding automaton becomes very complex and its generation too expensive, if not prohibitive. Since companies usually have a large number of business rules, the process execution may turn out to be impossible in many practical situations.

To tackle this problem, we propose a mechanism to verify the process rules at runtime. Our approach does not rely on generating all possible paths. Instead, we employ an efficient algorithm to block the execution paths that would lead to unacceptable behavior. For example, our algorithm prevents the process from reaching deadlock states. At the same time, we allow the execution of all activities that are valid according to the process business rules. Such strategy allows for the execution of larger business processes when compared to the automata-based approach.

2 Declarative Business Process

Declarative processes propose the use of *declarative languages* to define business rules that drive the business process execution [4]. This approach allows the process participants to take context-aware decisions during the process execution. On the other hand, one can define constraining business rules to prevent participants from performing prohibited or undesired activities.

The *DECLARE* framework, proposed by Pesic [4], is a tool for modeling and executing declarative processes. It offers rule templates that can be used to construct a process' business rules. These templates are mapped into formal expressions described in Linear Temporal Logic (LTL) and interpreted through a reasoning engine to identify enabled and prohibited activities during process execution.

ConDec, a graphical template language, is a constraint-based language developed by Pesic [4] for modeling business rules in declarative business processes. Its semantics is formally specified in LTL. Its graphical representation aims at improving its usability by non-LTL experts. Each constructor of the graphical language corresponds to a constraint template modeled by an LTL formula. These templates are classified in four groups, but for this paper only the following three are considered:

- **existential templates**: express the number of times an

activity can or needs to be executed in a process instance. This group comprises: *existence (A, N)* – indicates that an activity A must be executed at least N times; *absence (A, N)* – indicates that an activity A must be executed *at most* N times; and *init (A)* – indicates the first activity to be executed in the process.

- **relational templates**: express dependencies between activities. This group has five templates: *precedence (A, B)* – indicates that B cannot be executed before an activity A; *response (A, B)* – indicates that, for each execution of A, a certain activity B must be executed afterwards; *succession (A, B)* – corresponds to the conjunction of response and precedence; *coexistence (A, B)* – indicates that, if A is ever executed, certain activity B must also be executed and vice-versa; and *responded-existence (A, B)* – indicates that if A is ever executed, activity B must also be executed (either before or after A);

- **negation templates**: a negative version of the relational templates. For example, the template *not response (A, B)* indicates that after the execution of A, the activity B cannot be executed anymore; *not coexistence* indicates that after the execution of A, the activity B cannot be executed anymore, and vice-versa.

The analysis of LTL formulae rely on the generation of a Büchi automaton representing all acceptable traces that conform to the LTL formula [2]. The problem with this approach is that this automaton grows exponentially with model size and tends to become very large for complex models. Experiments we have conducted using ConDec showed that the number of states becomes too large even with a few dozen rules. In real business applications, a business process may require about thirty to fifty rules or more. This makes the use of ConDec impractical for most real business processes.

3 An approach for declarative processes that avoids the state space explosion problem

This paper proposes a mechanism to verify rules at runtime and control the execution of declarative processes. Our approach does not generate all possible execution paths.

Our proposed approach is called **ReFlex**. Once ReFlex starts the process execution, it interacts with the user to show the enabled activities at each execution point. ReFlex also warns the user when the process termination is enabled or disabled according to the pending activities. Figure 1 shows an overview of the proposed execution environment. During the modeling phase, the user specifies a business process in ConDec language (activities and constraints). This model is then compiled into a structure that stores the necessary information (activities states, counters, and rules fulfillment). ReFlex uses it to interpret the rules.

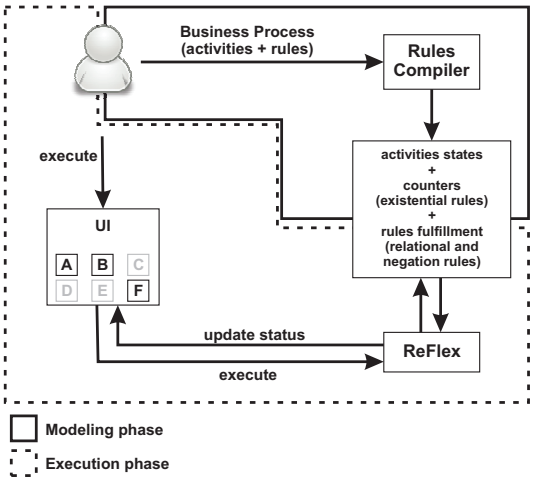

Figure 1. Overview of ReFlex.

In our approach, we adopt an additional state-based rule template that is not present in ConDec. The new rule is called "precedent-obliged (A, B)" and indicates that B cannot be executed while A is in the "obliged" state.

In the execution phase, ReFlex manages the structure generated by the compiler. This task involves changing activity states, and enabling/disabling the process termination. ReFlex is also responsible for interacting with the user interface. The interface notifies the user about activities enabled/disabled for execution. The user can only select enabled activities for execution. After executing an activity, ReFlex updates the structure and sends the new status to the user interface.

3.1 Problem representation

The concept of *activation* of a ConDec constraint was introduced by Burattin *et al.* [3]. The *activation* of a constraint in a trace is an event whose occurrence imposes some obligations on other events in the same trace. If the imposed obligation is fulfilled then the activation is considered a *fulfillment*; if the obligation is not fulfilled, it will be called a *violation*. Based on the concepts of *fulfillments* and *violations* introduced, Definition 3.1 shows how activities of a business process are represented in this work.

Definition 3.1 (Business process activities representation) *The activities of a process P are represented in a tuple $S_P = (\texttt{enabled}_P, \texttt{disabled}_P, \texttt{blocked}_P, \texttt{obliged}_P)$, where:*

- $\texttt{enabled}_P$ *is the set of enabled activities (the user is free to execute any activity in the set);*

- $\texttt{disabled}_P$ *is the set of disabled activities (the user cannot execute such activities at current execution point; these activities can became enabled in the future);*
- $\texttt{blocked}_P$ *is the set of blocked activities (activities that cannot be executed anymore). It is the set of activities that represent a violation to some rule in the process;*
- $\texttt{obliged}_P$ *is the set of obliged activities (activities that must be executed before the process termination). It is the set of activities that represent a fulfillment to some rule in the process.*

Throughout the process execution, a number of properties must remain valid. Definition 3.2 lists these properties.

Definition 3.2 (Properties of BP activities) *Let A be the set of activities in a process P.*

- $\forall a \in A, a \in \texttt{enabled}_P \cup \texttt{disabled}_P \cup \texttt{blocked}_P$*;*
- $\texttt{enabled}_P \cap \texttt{disabled}_P = \emptyset$*;*
- $\texttt{blocked}_P \cap \texttt{enabled}_P \cap \texttt{disabled}_P = \emptyset$*;*
- $\texttt{obliged}_P \subseteq \texttt{enabled}_P \cup \texttt{disabled}_P$

Burattin *et al.* also introduce the notion of *healthiness* of a trace. The *healthiness* of a process trace can be quantified based on the number of *activations*, and the number of *fulfillments*, and *violations* of these *activations*. A trace is "healthy" with respect to a constraint if the *fulfillment* ratio is 1 (one) and the *violation* ratio is 0 (zero).

The main concern of this work is to guarantee that the traces generated by the proposed approach are "healthy". In other words, we want to guarantee that all *activations* will be fulfilled and none of them will be violated.

Whenever an activity is executed, the activities states are updated to represent the process status. Table 1 describes how the activities states are updated according to the behavior of each type of rule.

During the compilation process, we identify triples (A, B, C) of activities that are inter-related such that A obliges C and B blocks C. These triples may cause deadlock if A and B are both executed before C. To avoid this situation, we apply the following *liveness-enforcing rule*.

Definition 3.3 (Liveness-enforcing rule) *For every triple (A, B, C) where A obliges C and B blocks C, include a new rule not_response(B, A) and a new rule precedent_obliged(C, B).*

Notice that the addition of these new rules may cause the emergence of new triples, which require the addition of new rules until all triples that may cause a deadlock have been handled.

Table 1. Behavior of each rule in the proposed approach.

Rules	Behavior
Existential Rules	
init(A)	All activities but A are disabled. After A is executed, the remaining rules determine the process status.
existence(A, n)	A is obliged until it is executed n times.
absence(A, n)	After $n-1$ executions of A, it is blocked.
exactly(A, n)	A is obliged until it is executed n times, but after n executions, it is blocked.
Relational Rules	
response(A, B)	After the execution of A, B is obliged.
precedence(A, B)	While A is not executed, B is disabled.
succession(A, B)	After the execution of A, B is obliged, but it is disabled while A is not executed.
coexistence(A, B)	If A is executed, B is obliged, and vice-versa (only for the first execution).
responded_existence(A, B)	The first execution of A obliges B.
Negation Rules	
not_response(A, B)	After the execution of A, B is blocked.
not_coexistence(A, B)	After the execution of A, B is blocked, and *vice-versa*.
State-Based Rules	
precedent_obliged(A, B)	While A is obliged, B is disabled.

4 Complexity of the proposed approach

In this section we analyze the memory requirements of our algorithm comparing it against the LTL-based approach.

Space complexity of an algorithm is defined in terms of what limits the number of tape cells a Turing Machine (TM) needs to use during its computation. Definition 4.1 presents the formal definition of deterministic space bounded computation [1].

Definition 4.1 (Space-bounded computation [1]) *Let $S : \mathbb{N} \rightarrow \mathbb{N}$, $s \in S$, and $L \subseteq \{0,1\}^*$. We say that $L \in$ **SPACE**$(s(n))$ if there is a constant c and a TM M deciding L such that at most $c \times s(n)$ locations on M's work tapes (excluding input tape) are ever visited by M's*

head during its computation on every input of length n.

Considering a TM with two tapes (a read-only input tape and a read/write work tape), on the read-only tape the input head can read symbols but not change them. The work tape may be read and written in the usual way. Only cells of work tape contribute to the space complexity in such a TM [6].

A complexity class is a set of problems of related complexity defined by factors such as: (i) the model of computation; and (ii) the resources that are being bounded. **L** is the class of problems that are decidable in logarithmic space on a deterministic Turing machine [6]. In other words, $L = SPACE(log\ n)$.

In our case, the input tape stores all activities and rules of a process. Let us say that this requires n cells. Looking at the declarative problem defined in Section 3, the work tape is used to store: (i) for each activity, its state; (ii) for each existential rule (except init), a single cell containing a counter for how many times the associated activity was executed; and (iii) for each relational and negation rule, a single cell with a mark indicating if the rule was fulfilled. Hence, the amount of data stored in the work tape is always smaller then the input tape. Rules require more than a single cell to be stored in the input tape, and, on the other hand, they require only one cell in the work tape. Figure 2 shows a representation of the TM that represents our approach. We consider that the existential counters are limited to a number less than infinite, so they can fit in a single cell if we use a sufficiently large tape alphabet.

Once the size of the work tape is always less than the input n, our algorithm is in the **L** complexity class.

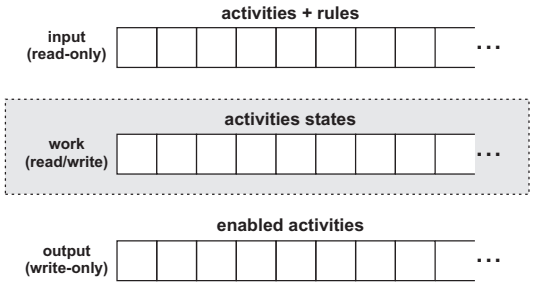

Figure 2. Space bounded computation for the proposed approach.

Comparing to the LTL approach, it is known that the complexity of model checking for LTL is in **PSPACE** complexity class [5]. **PSPACE** is the class of problems that are decidable in polynomial space on a deterministic TM. This is due to the state space explosion problem. As our approach does not share this problem, the problem is decidable in logarithmic space (**SPACE(log** n**)**).

5 Conclusions

In this paper, we approached the state space explosion problem in the context of declaraive business processes. LTL-based approaches require the computation of an automaton representing all acceptable traces. Unfortunately, this is often impracticable for most real business processes.

To tackle this problem, we proposed a mechanism that stores in activities and rules states all the information necessary to control the process. At the compilation process, a liveness-enforcing mechanism blocks the execution of paths that result in deadlocks, without affecting valid execution paths. Such mechanism disables or blocks activities at the necessary moments in order to prevent the user from mistakenly driving the process into unacceptable states. This strategy allows for the efficient execution of large business process models using the declarative paradigm.

We compared our approach to the traditional LTL-based approach, in terms of space complexity. While the LTL-based approach faces the state space explosion problem, which is an problem of **PSPACE** complexity, the mechanism proposed in this paper reduces the problem to the **L** class of complexity, which allows for an efficient implementation of a declarative business process engine.

References

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[2] Luboš Brim, Ivana Černá, Pavel Krčál, and Radek Peláek. Distributed ltl model checking based on negative cycle detection. In *Foundations of Software Technology and Theoretical Computer Science*. Springer Berlin / Heidelberg, 2001.

[3] Andrea Burattin, Fabrizio Maria Maggi, Wil M. P. van der Aalst, and Alessandro Sperduti. Techniques for a posteriori analysis of declarative processes. In *EDOC*, pages 41–50, 2012.

[4] Maja Pesic. *Constraint-Based Workflow Management Systems: Shifting Control to Users*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 2008.

[5] Philippe Schnoebelen. The complexity of temporal logic model checking. In *Proceedings of the 4th Workshop on Advances in Modal Logic (AIML'02)*, pages 481–517. King's College Publications, 2003.

[6] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 2nd edition, 2006.