Better Algorithms for Analyzing and Enacting Declarative Workflow Languages Using LTL

Michael Westergaard[*]

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
m.westergaard@tue.nl

Abstract. Declarative workflow languages are easy for humans to understand and use for specifications, but difficult for computers to check for consistency and use for enactment. Therefore, declarative languages need to be translated to something a computer can handle. One approach is to translate the declarative language to linear temporal logic (LTL), which can be translated to finite automata. While computers are very good at handling finite automata, the translation itself is often a road block as it may take time exponential in the size of the input. Here, we present algorithms for doing this translation much more efficiently (around a factor of 10,000 times faster and handling 10 times larger systems on a standard computer), making declarative specifications scale to realistic settings.

1 Introduction

Worflow languages provide an efficient means of describing complex workflows allowing analysis and assistance of users. Traditional workflow languages like, e.g., BPMN [12] require that the modeler explicitly specifies all possibilities from each state of the system, making it difficult to model more abstract relations between tasks when the user has many choices in each state. Descriptions such as "you are only young once; when you are young you should get an education; you can only get one master's degree; and only with a master's degree in business information systems will you truly be a master of business process modeling" can only with difficulty be implemented using a transitional workflow language and the complexity grows as the freedom increases. For this reason declarative workflow languages, such as Declare (also referred to as ConDec or DecSerFlow) [15] are becoming popular. Declarative languages do not explicitly state the allowed choices, but instead focus on constraints between tasks, and hence on disallowed behaviour rather than allowed behaviour. They are therefore well-suited to describing systems like the aforementioned example.

[*] This research is supported by the Technology Foundation STW, applied science division of NWO and the technology programme of the Dutch Ministry of Economic Affairs.

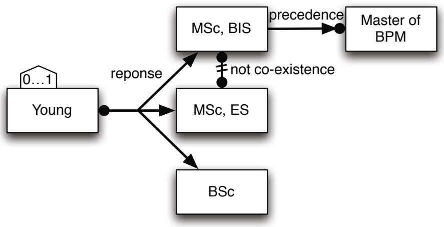

Fig. 1. A process described using Declare

Specifications in Declare consist of tasks, represented as rectangles, and constraints, represented as (hyper-) arcs between tasks. An example Declare specification is shown in Fig. 1. Here we have five tasks (e.g., Young) and four constraints. The constraint response from Young to MSc, BIS, MSc, ES, and Bsc states that if Young is executed, at least one of the others have to be executed to as well. The not co-existence constraint between the two MSc tasks state that at most one of these can be executed. Finally, the precedence from MSc, BIS to Master of BPM states that Master of BPM can only be executed if MSc, BIS has been executed previously. Finally, Young has a constraint stating that it can at most be executed once (the shape with 0..1 above the task). Thus, this is a Declare implementation of the example mentioned earlier. We shall not go into further details about the available constraints and their graphical representation here, but refer the interested reader to [15].

To provide analysis for and enact declarative languages, we translate specifications to a form that can be used by computers. For Declare, this has been done by translating specifications to the event calculus [2] and by translating specification to linear temporal logic (LTL) [15]. Here, we are concerned with translation to LTL as this allows more advanced analysis prior to execution, as LTL formulae can be translated to finite automata accepting the traces accepted by the LTL formula and original model [6, 7] Analysis made available by this is checking whether the specification is inconsistent, which manifests itself as an empty acceptance language and easily checked on a finite automaton. We can also identify tasks that can never be executed in a valid execution by inspecting the labels of the automaton, and identify redundant constraints by checking language inclusion using automata. Furthermore, an automaton is used for enactment of the workflow specification, by following states in the automaton and provide the user choices based on what the automaton allows.

The classical approach to translating LTL to automata [6] results in a Büchi-automaton accepting all infinite execution traces satisfying the formula. A slight modification of the algorithm [7] instead constructs a standard finite automaton accepting all finite traces accepted by the formula. This algorithm is intended for general model-checking, and thus allows requiring multiple atomic propositions to be satisfied at the same time. A problem with this algorithm is that the construction may be exponential in time and space. As our atomic propositions are events that cannot happen at the same time, [14] further improves on the base original algorithm by removing all transitions that require more than one event occurring at the same time, resulting in smaller automata and shorter execution time. The model-checking community has provisions for dealing with the size of automata by minimizing the formula prior to translation [4] or minimizing the generated automata on-the-fly or afterwards [13,4]. While these techniques alleviate the explosion to a certain degree, the model-checking community typically write formulae by hand which result in smaller formulae, and thus their problem is not so much difficulty in generating the automaton in the first place, but rather the subsequent analysis. We, however, generate formulae automatically from easier to understand descriptions, so our formulae become much larger, making reduction during the initial generation important.

Here, we improve on state-of-the art by exploiting characteristics of LTL formulae derived from declarative workflow specifications, namely that they are in essence huge conjunctions of relatively simple formulae. Traditional LTL translations handle conjunctions quite poorly, so we instead translate both sides of a conjunction individually and subsequently construct the synchronized product of the two automata, resulting in better performance. Having sub-divided the problem allows further improvement. For example, we can use automaton minimization algorithms prior to synchronizing, and we can order the product using heuristics to reduce intermediate products and achieve better performance. In this paper we describe several algorithms for constructing the automaton for a Declare model more efficiently. Most of our algorithms have a theoretical worst-case execution time no better that the standard algorithms (i.e., exponential in the size of the input), but behave much better in practise. One presented algorithm only uses linear time to construct an automaton which is useful but not the same as the one constructed by the traditional algorithm (it is a non-deterministic automaton for the negation of the model). We have implemented our algorithms in the Declare tool [3], and the improvements yields speed improvements of a factor 1,000-10,000 or more for randomly generated Declare models, and allows us to generate automata for descriptions consisting of in the order of 50 tasks and constraints, where previous algorithms only allowed us to generate automata for descriptions consisting of around 5-10 tasks and constraints. This in our view is a game-changer, as this makes it possible to represent real-life systems and not just toy-examples.

The rest of this paper is structured as follows: In Sect. 2, we introduce the concepts useful for understanding the details of the rest of the paper; this may be skipped on a first reading for the reader not interested in the discussions about theoretical run-time. In Sect. 3, we present algorithms for translating Declare specifications to finite automata using LTL, and in Sect. 4, we provide experimental results from our implementation of the improved algorithms. Finally, in Sect. 5 we sum up our conclusions and provide directions for further work.

2 Background

In this section we briefly introduce linear temporal logic (LTL) interpreted over finite traces, the specific kind of finite automata we are dealing with in this paper, the classical algorithm for translating LTL to finite automata including the improvements from [14], and the complexity of common operations on automata. This section can be skipped by a reader not interested in the asymptotic analysis provided. Most of the material presented in this section is known; the contribution here is the formalization of the non-standard kind of finite automaton we use (Def. 2) and single-event automata (Def. 3), and the introduction of a notion of a deterministic automaton for our kind of finite automaton (Def. 4).

In the following, we interpret formulae and automata over finite sequences, $\sigma = e_0e_1e_2\dots e_{n-1}$ for $e_i \in E$. We call the set of all such sequences E^* and use the notation $\sigma(i) = e_i$, $\sigma\langle i\rangle = e_ie_{i+1}\dots e_{n-1}$, and $|\sigma| = |e_0e_1\dots e_{n-1}| = n$. The concatenation of two sequences $\sigma_1 = e_{1,0}e_{1,1}\dots e_{1,n-1}$ and $\sigma_2 = e_{2,0}e_{2,1}\dots e_{2,m-1}$ is $\sigma_1\sigma_2 = e_{1,0}e_{1,1},\dots e_{1,n-1}e_{2,0}e_{2,1}\dots e_{2,m-1}$.

Definition 1 (LTL Syntax). *Given a set, AP, of **atomic propositions**, a formula φ of **linear temporal logic (LTL)** over a set of atomic propositions AP has the form*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid X\varphi \mid \varphi U \psi$$

where $p \in AP$, and φ and ψ are LTL formulae. We allow the abbreviations $\varphi \implies \psi = \neg\varphi \vee \psi$, $F\varphi = \top U \varphi$ and $G\varphi = \neg F(\neg\varphi)$. □

We interpret LTL over finite sequences by assuming a labeling function: $\lambda : S \to 2^{AP}$. The standard logical connectives behave as normal, i.e., $\sigma \models p$ iff $p \in \lambda(\sigma(0))$, $\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$, $\sigma \models \varphi \vee \psi$ iff $\sigma \models \varphi$ or $\sigma \models \psi$, and $\sigma \models \varphi \wedge \psi$ iff $\sigma \models \varphi$ and $\sigma \models \psi$. $X\varphi$ means that φ must held in the next state, i.e., $\sigma \models X\varphi$ iff $\sigma\langle 1\rangle \models \varphi$[1], and $\varphi U \psi$ means that φ holds until ψ holds and ψ holds eventually, i.e., $\sigma \models \varphi U \psi$ iff $\exists k.0 \leq k < |\sigma|$ such that $\sigma\langle k\rangle \models \psi$ and $\forall i.0 \leq i < k$ it holds that $\sigma\langle i\rangle \models \varphi$. We call the set of all sequences satisfying a formula φ the *language* of φ, denoted $L(\varphi) = \{\sigma | \sigma \models \varphi\}$.

Our definition of automata is very similar to the traditional one, except we add a little more structure to the labels. Labels are a set of atomic propositions or negated atomic propositions. Intuitively, in order to transition from one state to another, one must satisfy all positive atomic propositions and none of the negative atomic propositions of a label. An empty set of labels indicates a transition that can be followed regardless of the set of atomic satisfied.

Definition 2 (Finite Automaton). *A **finite automaton** FA is a tuple $FA = (L, S, \delta, s_I, A)$, where L is a finite set of **labels**, S is a finite set of **states**, $\delta \subseteq S \times 2^{L \cup \neg L} \times S$ is the **transition relation** where $\neg L = \{\neg p | p \in L\}$, $s_I \in S$ is the **initial state**, and $A \subseteq S$ is the set of **accepting states**.* □

We say a sequence $\sigma = s_0s_1\dots s_{n-1} \in E^*$ with a labeling function $\lambda : E \to AP$ is accepted by a finite automaton $FA = (L, S, \delta, s_I, A)$ iff $\forall i.0 < i < n$ there exists a transition, t, leading from s_{i-1} to s_i, i.e., $(s_{i-1}, t, s_i) \in \delta$, such that $s_0 = s_I$ and the labels of s_{i-1} are consistent with t, i.e., $t \cap L \subseteq \lambda(e_{i-1})$ and $\neg\lambda(e_{i-1}) \cap t = \emptyset$, and the trace ends up in an accepting state, i.e., $s_{|\sigma|-1} \in A$.

[1] There are some intricacies at the end of the trace leading to introduction of two kinds of next operators [11]; they are not material in this paper and is discussed in a bit more detail when we talk about accept states for generated automata.

We call the set of all sequences accepted by a finite automaton the *language* of FA, denoted by $L(FA) = \{\sigma \in E^* | \sigma \text{ is accepted by } FA\}$. Note that we make no assumptions about $L = AP$.

The standard algorithm for translating LTL to finite automata defines and builds the automaton inductively. Each state has a proof obligation, a set of formulae that must be satisfied in order for a trace starting in that state to be satisfied. The algorithm starts by bringing the formula to *negative normal form*, i.e., a form where negations only appear directly in front of atomic propositions. This is done by using De Morgan's laws, i.e., $\neg(A \vee B) = \neg A \wedge \neg B$ and $\neg(A \wedge B) = \neg A \vee \neg B$, and introducing the release operator, $\varphi V \psi$ defined as $\neg(\varphi U \psi) = \neg\varphi V \neg\psi$. Given a formula in negative normal form, φ, the algorithm starts with a single state with proof obligation φ. This state is furthermore marked as initial. Now, proof obligations are split up according to the structure of φ. If $\varphi = \psi \wedge \rho$, we add ψ and ρ as new proof obligations, and mark φ as satisfied. If $\varphi = \psi \vee \rho$, we duplicate the state (including any transitions) and add ψ to one copy and ρ to the other, and mark φ as satisfied. For $\varphi = X\psi$, we add a new state with the proof obligation ψ and add a transition from the current state to the new state. The intuition is that we consume a single event and in the next state we must satisfy ψ. $\varphi = \psi U \rho$ and $\varphi = \psi V \rho$ are both handled using their fix-point characterisation, namely that $\psi U \rho = \rho \vee (\psi \wedge X(\psi U \rho))$ and $\psi V \rho = \rho \wedge (\psi \vee X(\psi V \rho))$, and using the other rules. We make sure only to create one state for each set of proof obligations, reusing previously created states if possible. We do not need to handle the case where $\varphi = \neg\psi$ as this only happens when $\psi = p$ for some $p \in AP$.

To define a finite automaton $FA_\varphi = (L, S, \delta, s_I, A)$, the set of states, S, and the initial state s_I are as defined by the inductive algorithm. L coincides with the atomic propositions, AP, of φ. A state can transition from one state to another if we have created a transition as explained in the inductive algorithm and the labels of the transition are atomic propositions and negated atomic propositions of the source, i.e., $(s, t, s') \in \delta$ iff s and s' are related as described above and $t = s \cap 2^{AP \cup \neg AP}$. The accepting states are all states with no outstanding temporal obligations. For V this does not have any impact, and for U it just means that a state cannot have any U proof obligation (an obligation $\psi U \rho$ has not yet satisfied ρ, and hence cannot be accepted). For X the problem is a bit harder, and one has to split X up into a weak and a strong variant [9]. The weak variant basically means "if there is a next state, it must satisfy this formula" and the strong variant means "there is a next state and it satisfies this formula", and is hence an outstanding proof obligation. Except at the end of the trace, the two versions behave the same. We have that $L(\varphi) = L(FA_\varphi)$.

To improve the algorithm one typically checks each state for inconsistencies before expanding it. A state is inconsistent if it contains both a formula, φ, and its negation, $\neg\varphi$. All inconsistent states are removed. In order to improve the algorithm for workflow specifications, [14] extends this requirement by adding the additional condition that a state is inconsistent if it contains two atomic propositions, $p \neq q$ (as a sequence from a workflow always produces exactly one event so two atomic propositions can never be satisfied at the same time). Furthermore, whenever a state contains an un-negated atomic proposition, p, we can remove all negated atomic propositions (except $\neg p$). This may allow us to merge the state with others as it contains fewer proof obligations and thus may match another state. Thus, an automaton constructed using this approach will only have transitions that are either a single positive label or a (possibly empty) set of negative labels. We call such an automaton a *single-event finite automaton* as formalised in the following definition.

Definition 3 (Single-event Finite Automaton). *A finite automaton $FA = (L, S, \delta, s_I, A)$ is said to be a* ***single-event finite automaton*** *iff for all t such that $(s_{i-1}, t, s_i) \in \delta$ it holds that either $t = \{p\}$ for some $p \in L$ or $t \subseteq \neg L$ (note that this includes $\emptyset$).* □

All automata we have used until now are non-deterministic, i.e., whenever we are given a state, s, and a transition, t, there is not necessarily a unique successor state, s', such that $(s, t, s') \in \delta$ and $(s, t, s'') \in \delta \implies s' = s''$. We define a *deterministic single-event finite automaton* as an automaton where this holds:

Definition 4 (Deterministic Single-event Finite Automaton). *A single-event finite automaton $DFA = (L, S, \delta, s_I, A)$ is said to be* ***deterministic*** *iff for all $s \in S$, either*

- *there exists a $s' \in S$ such that $(s, \emptyset, s') \in \delta$ and $(s, t, s'') \in \delta \implies s'' = s' \wedge t = \emptyset$, or all of the following hold*
- *for all $p \in L$ there exists a $s'_p \in S$ such that $(s, \{p\}, s'_p) \in \delta$ and $(s, \{p\}, s''_p) \in \delta \implies s''_p = s'_p$,*
- *there exists a s' such that $(s, \neg L, s') \in \delta$ and $(s, \neg L, s'') \in \delta \implies s'' = s'$, and*
- *if $(s, t, s'_t) \in \delta$ either $t = \{p\}$ for some $p \in L$ or $t = \neg L$.* □

Given two finite automata sharing labels, FA_1 and FA_2, we can construct an automaton $FA_1 \times FA_2$ accepting the intersection of the languages accepted by the original automata, i.e., such that $L(FA_1) \cap L(FA_2) = L(FA_1 \times FA_2)$. Due to the semantics of LTL, we have $L(FA_{\varphi \wedge \psi}) = L(\varphi \wedge \psi) = L(\varphi) \cap L(\psi) = L(FA_\varphi) \cap L(FA_\psi) = L(FA_\varphi \times FA_\psi)$. We can construct this product using the *product construction*, which carries deterministic automata over to deterministic automata and uses time and space $O(|S_1||S_2| \cdot 2^{2|L|})$ if the automata are non-deterministic (a non-deterministic automaton can have up to $2^{2|L|}$ different transition labels) and $O(|S_1||S_2||L|)$ for deterministic automata (they have a restricted set of labels). We can construct an automaton $FA_1 + FA_2$ accepting the union of the original languages, i.e., $L(FA_1) \cup L(FA_2) = L(FA_1 + FA_2)$, using the similar *sum construction*. If the resulting automaton does not have to be deterministic, we can construct an automaton accepting the same language by setting the states as the disjoint union of the original sets of states, make the transition relation respect the original transition relations, and make accepting states of the sum be the disjoint union of the original accepting states.

The initial state is a new state which has transitions that are the union of the transitions from the original initial states. This construction only uses time and space $O(|S_1| + |S_2| + 2^{2|L|}$. It is possible to minimise a finite automaton, deterministic or not, i.e., given a finite automaton, FA, to find another automaton, FA', such that $L(FA) = L(FA')$ but such that FA' contains the minimum number of states possible to accept $L(FA)$. Basically, one can try all automata smaller than a given automaton and pick one with the fewest number of states. For non-deterministic automata, this is the best possible algorithm in the worst case, but for deterministic automata, it is possible to do this more efficiently, and obtain a (unique up to renaming) minimal automaton in time $(O(|L| \cdot |S| \log |S|))$ [8]. For every finite automaton $FA = (L, S, \delta, s_I, A)$, there exists a deterministic finite automaton, $DFA_{FA} = (L, S', \delta', s'_I, A')$, such that they accept the same language, i.e., $L(FA) = L(DFA_{FA})$. The automaton can be constructed using the *subset construction* which uses time and space $O(2^{|S|} 2^{2|L|})$ (as we can have up to $4^{2|L|}$ different transitions). Given a deterministic automaton, FA, we can construct an automaton, FA^c accepting the complement, i.e., $L(FA)^c = L(FA^c)$, interpreted over some domain. Due to the semantics of LTL, we have $L(\neg\varphi) = L(\varphi)^c = L(FA_\varphi)^c = L(FA^c)$. Finding the complement of a non-deterministic automaton is as hard as constructing a corresponding deterministic automaton and negating that. This takes time and space $O(|S| + |L|)$ for deterministic automata. For any finite automaton, we can construct an automaton accepting the set of all prefixes of $L(FA)$, $\mathit{prefix}(L(FA)) = \{\sigma | \sigma\tau \in L(FA)\}$ for non-deterministic automata in time $O(|S| + 2^{2|L|})$ and for deterministic automata in time $O(|S| + |L|)$.

3 Algorithm

Here we describe various ways to construct an automaton accepting the language of a Declare workflow specification or its complement. Our implementation assumes that single-event automata are used, but the algorithms can be generalized to handle arbitrary automata. Declare allows users to specify workflows by describing a set of tasks, $T = \{T_i\}_{i=1}^m$, and a set of constraints, $C = \{C_i\}_{i=1}^n$, which are LTL formulae with atomic propositions $AP = T$. For the example in Fig. 1, we have $T = \{\mathsf{Young}, \mathsf{MScBIS}, \mathsf{MScES}, \mathsf{BSc}, \mathsf{MasterOfBPM}\}$ with $m = 5$ and $C = \{\neg(F(\mathsf{Young} \wedge X(F(\mathsf{Young})))), G(\mathsf{Young} \implies F(\mathsf{MScBIS} \vee \mathsf{MScES} \vee \mathsf{BSc})), \neg(F\mathsf{MScBIS} \wedge F\mathsf{MScES}), \neg((\mathsf{Master} U \mathsf{MScBIS}) \vee G\mathsf{Master})\}$ with $n = 4$ (the constraints are in order: “you are only young once”, “when you are young you should get an education”, “you can only get one master’s degree”, and “only with a master’s degree in business information systems will you truly be a master of business process modeling”). The allowed behaviour of the specification is the language accepted by $\varphi = \bigwedge_{i=1}^n C_i$.

In Declare constraints are picked from a set of templates, which are instantiated to concrete atomic formulae. Thus, the set of possible formulae for constraints is fixed and finite. It is easy to see that, given a formula, φ, if r is a bijective renaming of atomic propositions, the language of the formula obtained by renaming atomic propositions of the original formula, φ/r, is the same as the one obtained by taking an automaton accepting the language of the original formula, FA_φ, and renaming all labels of the automaton according to the renaming function, FA_φ/r, i.e., $L(\varphi/r) = L(FA_{\varphi/r}) = L(FA_\varphi/r)$. Thus, we can pre-compute the automata for all constraint templates and construct an automaton accepting $L(C_i)$, FA_{C_i}, in constant time. Furthermore, as the templates are known beforehand, the size of the entire specification $|\varphi| = |\bigwedge_{i=1}^n C_i| \leq n \cdot \max_{i=1}^n |C_n| \in O(n)$ as $\max_{i=1}^n |C_i| \in O(1)$. We only use this for arguing about the speed; our implementation computes an automaton for each constraint on-the-fly. This is linear in the size of the model as long as we stick to a pre-defined set of constraints.

A summary of the different algorithms developed in this paper can be seen in Table 1. For each algorithm, we give the worst case execution time; here $|\varphi|$ is the length of the formula $\varphi = \bigwedge_{i=1}^n C_n$, $n \in \theta(|\varphi|)$ is the number of constraints in the input and m is the number of tasks, $|S|$ is the maximum number of states of a automaton for any of the possible constraints and $|S'|$ is the maximum number of states in a deterministic such automaton. After renaming of labels in the template automata, we can consider them to range over the same set of labels, which corresponds to the tasks, so $|L| = m$. The memory requirement is the same as the time requirement in all cases. We see that the improved algorithms rarely provide better bounds than the base algorithm, except for algorithm 4, which provides a less valuable result. In the next section we turn to an experimental evaluation of the algorithms showing significant improvement in practise.

Base Algorithm. The base algorithm is using the standard translation with single-event optimisation on φ. The time for this algorithm is $O(2^{|\varphi|})$.

Algorithm 1: Construct automaton for the negation. The first idea is, instead of constructing an automaton for the specification $\varphi = \bigwedge_{i=1}^n C_n$, to construct one for the negation, $\neg\varphi$. The idea is that when translating $\neg\varphi$ to negative normal form, we get $\neg\varphi = \neg\left(\bigwedge_{i=1}^n C_n\right) = \bigvee_{i=1}^n(\neg C_n)$ using De Morgan's laws. While LTL translators are bad at handling conjunctions, as they just add proof obligations, they are good at handle disjunctions as they sub-divide the problem. This algorithm also runs in time $O(2^{|\varphi|})$.

Table 1. Comparison of the algorithms

Algorithm	Execution time	
	Non-deterministic	Deterministic
Base	$O(2^{\lvert\varphi\rvert})$	-
1	$O(2^{\lvert\varphi\rvert})$	-
2	$O(\lvert S\rvert^n 2^{2m})$	$O(\lvert S'\rvert^n m)$
3	$O(2^{\lvert S\rvert^n} 2^{2m} n)$	$O(\lvert S'\rvert^n mn^2 \log \lvert S'\rvert)$
4	$O(n)$	same as algorithm 2 or 3
5,6,8	same as algorithm 2 or 3	same as algorithm 2 or 3
7	same as algorithm 2 or 3 for each partition	same as algorithm 2 or 3 for each partition

Algorithm 2: Use automaton operations to handle conjunctions. The results of using algorithm 1 show significant speed improvement compared to the base algorithm, and suggest that runtime is dominated by the difficulty in efficiently coping with conjunctions. Unfortunately, we often want to construct the prefix automaton for a Declare specification, but this cannot be done efficiently from a non-deterministic automaton for the negated model. We thus seek an automaton accepting $L(\varphi)$with as many of the nice traits of algorithm 1 as possible. One way to achieve our goal is to instead construct the automaton using $L(\varphi) = L\left(\bigwedge_{i=1}^{n} C_n\right) = L\left(\prod_{i=1}^{n} FA_{C_i}\right)$ where $\prod_{i=1}^{n} FA_{C_i}$ is shorthand for $FA_{C_1} \times FA_{C_2} \times \cdots \times FA_{C_n}$. As we have pre-computed FA_{C_i}, time spent in this algorithm is just the time for constructing the product, which is $O(|S|^n 2^{2m})$ for non-deterministic automata and $O(|S'|^n m)$ for deterministic automata. While $|S'| \in O(2^{|S|})$, the number of potential labels are smaller ($|m|$ vs 2^{2m}) and in practise they are rarely larger. Though $n \in \theta(|\varphi|)$, it is numerically smaller.

Algorithm 3: Use minimisation. One way to make the base of the runtime of algorithm 2 smaller in practise is to minimize the sub-automata used in the computation of $\prod_{i=1}^{n} FA_{C_i}$. For non-deterministic automata this takes $O\left(2^{|S|^n} 2^{2m} n\right)$ (n times minimization of at most $|S|^n$ states, each dominating the multiplication producing the automaton). For deterministic automata, this takes $O\left(|S'|^n m n^2 \log |S'|\right)$ (same reason as in the non-deterministic case, but with a different complexity for minimization). While this theoretically is worse, the hope is that using smaller automata improves performance in practise.

Algorithm 4: Using automata to compute the negation. Even though the negation cannot be used to compute the prefix automaton directly, we can still use it to find definite violations, and it is thus interesting to find efficient algorithms for it. Using the results from algorithm 2 combined with algorithm 1, we can obtain a non-deterministic automaton in time $O(n|S|2^{2m})$ we just have to modify initial states n times, every time taking time proportional to the number of out-going transitions. We note that $|S|$ is a small constant only depending on the constraints and 2^n is an over-approximation as the total number of edges is also independent of the model and known from the constraint automata, in effect yielding a running time of $O(n)$. For deterministic automata, we have to use a product construction, leading to the same time as for algorithm 2 or 3.

Algorithm 5: Using balanced trees. The commutative law applies for conjunctions of LTL formulae, so we may re-arrange the conjunction prior to translation. This allows us to arrange the conjunctions in a balanced tree as shown in Fig. 2 (bottom) instead of sequentially as shown in Fig. 2 (top). Arranging the automata in a tree can done in linear time, and the time for computing the product is theoretically the same as for algorithm 3 (as this is just a special-case). We do not expect it to perform significantly better or worse than the sequential implementation used for algorithm 3.

Algorithm 6: Using automaton size. Looking at the tree arrangement in Fig. 2 (bottom), we have more freedom when it comes to the size of the intermediate products (the diamonds).The reason is that we can compute half the intermediate products independently of other computations. We want to keep the size of intermediate products as small as possible to reduce the time spent computing them (the time spent is proportional to the size of the result). One heuristics to do that is to ensure we never compute the product of two large automata. The idea of algorithm 6 is to construct a tree like in algorithm 5 and order leaves so large automata are paired with small ones. On the next levels we do the same.

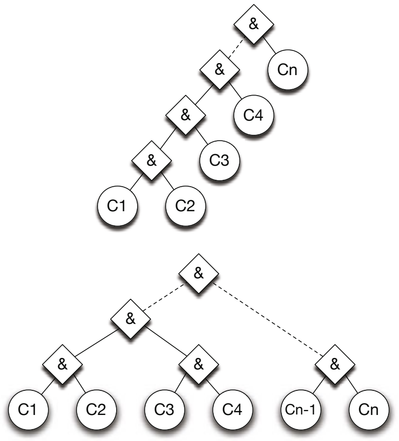

Fig. 2. Conjunction arrangements

Algorithm 7: Computing partial product only. In order to detect inconsistencies in a Declare model, we check whether the language of the derived LTL formula is empty, i.e., if $L(\varphi) = \emptyset$. If we can find C_A and C_B with $\varphi = C_A \wedge C_B$ such that $L(\varphi) = \emptyset \iff L(C_A) = \emptyset \vee L(C_B) = \emptyset$ we can just check the (possibly) smaller automata for C_A and C_B for emptiness. As we have $\varphi = \bigwedge_{i=1}^{n} C_n$, we seek $P \subseteq \{1, 2, \ldots, n\}$ so $C_A = \bigwedge_{i \in P} C_i$ and $C_B = \bigwedge_{j \in \{1,2,\ldots,n\} \setminus P} C_j$. Even though automata for C_A and C_B may not be smaller than the automaton for φ, at least we can construct them easier (as they consist of fewer conjunctions).

In terms of languages we seek C_A and C_B such that $L(C_A) \cap L(C_B) = L(C_A \wedge C_B) = \emptyset \iff L(C_A) = \emptyset \vee L(C_B) = \emptyset$. Let us consider the contraposition of the direction we are interested in, left to right, namely $\neg (L(C_A) = \emptyset \vee L(C_B) = \emptyset) \iff L(C_A) \neq \emptyset \wedge L(C_B) \neq \emptyset \implies L(C_A) \cap L(C_B) \neq \emptyset$. We must construct C_A and C_B such that if we have a $a \in L(C_A)$ and a $b \in L(C_B)$ then we have a $c \in L(C_A) \wedge c \in L(C_B)$. We construct C_A and C_B such that $c = ab$ can be used. If C_A is closed under adding suffixes, i.e., if $a \in L(C_A)$ then $aw \in L(C_A)$ for any word $w \in \Sigma^*$, we clearly have that $c = ab \in L(C_A)$. If furthermore C_B is closed under adding prefixes, i.e., if $b \in L(C_B)$ then $wb \in L(C_B)$ for all words $w \in \Sigma^*$, we also clearly have that $c = ab \in L(C_B)$.

Imposing this notion of suffix/prefix closedness is a too strong condition, however, as we would be able to do anything in the suffix/prefix. Instead, we seek a weaker notion that preserve the property. The idea is that constraints from Declare typically have some part that activate them and some part that subsequently satisfy them. The constraints do not care what happens when they are not activated. In the example in Fig. 1, the constraint response is activated by the execution of Young and satisfied by the execution of one of the MScs or BSc. Whether the constraint is satisfied is not affected by any other tasks. Some constraints are activated in the initial state, e.g., the 0..1 constraint on Young.

For a partitioning into C_A and C_B we are satisfied if we can pick a suffix/prefix to add to a string of the language accepted by one to the language of the other. Assume that C_A is an automaton derived from a constraint with formula φ. We then denote by $AP(C_A)$ the set of all atomic propositions occurring in φ. By construction, this coincides with the labels occurring in C_A. If $a \in L(C_A) \implies av \in L(C_A)$ for any $v \notin AP(C_A)$, we conclude that for any string $w \in (AP(C_A)^c)^*$ $a \in L(C_A) \implies aw \in L(C_A)$. For the same reason, if $b \in L(C_B) \implies vb \in L(C_B)$ for $v \notin AP(C_B)$ then $b \in L(C_B) \implies wb \in L(B)$ for any string $w \in (AP(C_B)^c)^*$.

If we have two automata, C_A and C_B, with $AP(C_A) \cap AP(C_B) = \emptyset$ then and an $a \in L(C_A)$ we can find another $a' \in L(C_A) \cap (AP(C_B)^c)^*$ which does not use any labels from $AP(C_B)$, as labels of $AP(C_B)$ does not occur in C_A. This is possible as C_A cannot distinguish characters not in $AP(C_A)$. Similarly, for a $b \in L(C_B)$ we can find a $b' \in L(C_B) \cap (AP(C_A)^c)^*$. Thus, $a'b' \in L(C_A)$ and $a'b' \in L(C_B)$, i.e., $a'b' \in L(C_A) \cap L(C_B) = L(C_A \times C_B)$ as wanted.

We thus need two automata C_A and C_B not sharing labels; one must satisfy $a \in L(C_A) \implies av \in L(C_A)$ for any $v \notin AP(C_A)$ and the other must satisfy $b \in L(C_B) \implies vb \in L(C_B)$ for any $v \notin AP(C_B)$. The first is easily checked by inspecting the labels of two automata and the latter can be identified by looking for self-loops on accepting and initial states, respectively. Both of these properties are preserved by automaton product, and any automaton derived from a constraint in Declare satisfies the requirement of accepting any suffix not in their own labels (when not enabled, they do not care what happens to tasks unconnected tasks). Most additionally satisfy the requirement for accepting any prefix not in their alphabet (all constraints not initially enabled).

This gives us the following partitioning algorithm: partition the graph with constraints as nodes and tasks as edges into connected components, considering not accepting any prefix as an additional constraint. This makes sure that no two automata from different partitions share labels and hence that the product of all automata from one partition does not share labels with the product of all automata from another. Furthermore, the automata derived from all partitions accept any suffix using different labels, and, except for at most one, all accept any prefix. Thus, the language of the product of any two products of all automata from two partitions is empty if and only if one of the factors is, and hence the product of all automata in all partitions is empty if and only if the product of all automata of at least one of the partitions is. We can construct the partitioning in linear time in the size of the original model and compute the product for each partitioning in the same time as for the other tree-based algorithms. This is an improvement as, if we have more than one, the partitions are smaller than the original system, thus reducing the exponent n in the running time. Automata for partitions can be calculated using algorithms 2, 3, 5, 6, and 8.

Algorithm 8: Taking atomic propositions into account. The idea of this algorithm is that synchronized products become smaller the more synchronisation takes place. In the previous section we have argued that partitioning according to connected components of the graph induced by constraints as nodes

and tasks as edges is a good idea. This graph typically contains few connected components (few tasks of the same specification that are not related by a chain of constraints), but it often contains islands that are only connected by few tasks. The idea of algorithm 8 is to use this graph for constructing the conjunction tree. We want to partition the graph into two parts, representing the left and right subtree, so as few atomic propositions as possible are shared between the two. This is equivalent to performing a minimum bisection or sparsest cut [5] and thus NP-hard. As we just use this as a heuristics for partitioning the constraints to construct a conjunction tree, we do not need a guaranteed best solution, so we use a simple hill-climbing implementation working from random partitions to approximate the problem. Aside from the improvement arising from constructing smaller intermediate products, we also get another benefit, as inconsistencies manifest them selves by incompatible requirements. Thus, by grouping constraints more likely to violate each other, we expect to find inconsistencies faster.

4 Experimental Validation

In this section we present results from experiments with the algorithms from the previous section. Unfortunately, we do not have real-life models large enough for our optimizations to really matter – for the most part because the base algorithm was unable to handle that – making it impossible to conduct interesting case studies beyond toy-examples. For this reason, we have experimented with randomly generated models, generated by adding constraints with equal probability and assigning tasks randomly. We make sure not to add constraints that are obviously in conflict (like forcing two tasks to be initial).

Completely randomly generated models are not interesting when they grow large. This is mostly because a large model is difficult to make satisfiable. We expect real-life models to be satisfiable or nearly satisfiable (i.e., by removing a few constraints, they become satisfiable). For this reason, we have decided to focus our evaluation on satisfiable models. We have made such models by randomly generating models and testing them using all our algorithms, marking them as satisfiable if any of the algorithms did. Naturally, this way of generating models is not optimal, as it ensures that we only test our algorithms on models that can be analyzed, but we still find this is better than using random models. We also check completely random models to ensure that this method of testing does not impose too much bias towards our algorithms.

We note that while the number of tasks, m, appear in the complexity of most of the algorithms in the previous section, it does not manifest in the same way in reality. The reason is that what is really interesting is the number of transitions in the resulting automaton, and adding more tasks that are not connected to any constraint does not add any more. Experiments show that the complexity tops when the number of tasks is same as the number of constraints or slightly larger. As most of our toy-examples have slightly more tasks than constraints (case in point, Fig. 1), we have chosen to only show such configurations.

In Table. 2, we see the performance of each algorithm on a set of randomly generated models. All experiments are run with known satisfiable models, known unsatisfiable models, and random models. The first column indicates the number of tasks (T) and constraints (C) for each model. For each algorithm, we show the time spent for each model in milliseconds and the percentage of test-cases that could complete successfully (in parentheses; omitted if 100). We have indicated whether numbers result from running the non-deterministic or deterministic version of each algorithm by adding an n or a d after the algorithm number. We have run analysis for 120 random models in each case, allowing the algorithms to use up to 256 MiB of memory. To see the performance when granting algorithms more memory, we have also included numbers for the base algorithm and algorithms 7 and 8 allowing them to use 4 GiB memory on a slightly slower computer. These are indicated by adding a prime after the name (e.g., Base'). For some executions we have indicated that the average execution time is a minimum. This is because we have chosen to limit the execution for each model to 5 hours; when a series of experiments have one or more instances being terminated due to running out of time instead of running out of memory, we have included it in the time average but not in the completion percentage, and we have indicated that the time is a lower bound.

For satisfiable models, the base algorithm only copes for the smallest examples. Negating the property (algorithm 1) makes it handle all cases. We see the same by computing the non-deterministic automaton for the negation (algorithm 4n). Using automaton properties (algorithm 2) does not improve on the base algorithm in the non-deterministic case, as we end up computing products with a large number of transitions. For deterministic automata, this approach fares much better, however. Especially when we combine this approach with minimization of intermediate automata (algorithm 3). Organizing the products in a tree (algorithm 5), yields faster results but handles slightly fewer cases than just computing the product (algorithm 3) as the intermediate products are smaller requires slightly more memory as we have to store more of them. Being intelligent about the organization of the tree yields an improvement, as shown by ordering by size (algorithm 6) and especially when ordering by shared atomic propositions (algorithm 8). The largest contribution is computing partial products only (algorithm 7). We have used algorithm 7 with the grouping of trees in each factor using algorithm 8. While most algorithms impose a small penalty on adding more tasks when keeping the number of constraints constant, algorithm 7 actually does the opposite. The reason is that more tasks makes it possible to have more factors in randomly generated models.

We note that most algorithms fare better for unsatisfiable models than for satisfiable models as they all have a notion of early termination as soon as a formula is recognized as unsatisfiable. This case is not expected to occur commonly in practise as human-constructed models are constructed to be consistent. The algorithms grouping related constraints (7 and 8) fare particularly well, supporting our expectation that they find contradictions earlier.

Table 2. Experimental results

	T,C	Base		Base'		1	2n		2d		3d		4n
		Time	(%)	Time	(%)	Time	Time	(%)	Time	(%)	Time	(%)	Time
satisfiable	10,5	35		55		1	17		2		2		1
	15,10	9172	(83)	4.82e4		1	1249	(7)	71		70		1
	20,15	1.28e4	(11)	> 6.06e6	(48)	1			1096	(76)	1093	(85)	1
	30,15	1.31e4	(3)	> 1.09e7	(26)	2			2117	(39)	2116	(65)	2
	30,20			> 1.67e7	(5)	2			3748	(5)	3365	(33)	2
	50,30			> 1.57e7	(10)	3			511	(1)	2666	(2)	3
	70,50					6							6
unsatisfiable	10,5	4		19		1	3		2		1		1
	15,10	1791	(98)	1.15e4		1	435	(42)	18		14		1
	20,15	> 2.36e5	(64)	> 2.11e6	(86)	2	48	(13)	379	(91)	222	(98)	2
	30,15	1650	(53)	> 3.32e6	(74)	2	32	(19)	1328	(76)	595	(93)	2
	30,20	6424	(58)	> 5.53e6	(61)	2	89	(12)	463	(35)	1057	(82)	2
	50,30	> 3.22e6	(50)	> 5.12e6	(64)	3	221	(13)	196	(26)	1289	(40)	3
	70,50			> 1.59e7	(12)	7	669	(9)	69	(15)	377	(21)	6
random	10,5	51		83		1	10		3		2		1
	15,10	> 3.60e5	(83)	7.91e4		1	471	(17)	64		62		1
	20,15	> 3.66e5	(41)	> 3.93e6	(62)	1	49	(10)	776	(89)	650	(89)	1
	30,15	2458	(19)	> 8.97e6	(36)	2	2	(3)	1889	(68)	1278	(68)	2
	30,20	> 3.53e6	(31)	> 1.17e7	(32)	2	112	(6)	916	(42)	1537	(42)	2
	50,30	> 7.88e6	(26)	> 1.10e7	(31)	3	91	(7)	113	(16)	1018	(21)	3
	70,50					7	89	(3)	235	(8)	270	(13)	7

	T,C	5d		6d		7d		7d'		8d		8d'	
		Time	(%)	Time	(%)	Time	(%)	Time	(%)	Time	(%)	Time	(%)
satisfiable	10,5	3		2		2		7		2		3	
	15,10	46		45		6		9		31		44	
	20,15	692	(86)	718	(88)	27		49		365	(90)	644	
	30,15	1314	(60)	1144	(63)	9		20		728	(71)	1763	
	30,20	4120	(21)	3313	(28)	152	(99)	261		1262	(43)	4135	
	50,30					210	(83)	2385		3040	(8)	4768	(12)
	70,50					832	(54)	4512					
unsatisfiable	10,5	1		2		11		4		2		1	
	15,10	8		10		9		7		4		5	
	20,15	294		109	(99)	9		14		21		24	
	30,15	266	(94)	315	(95)	6		11		63	(97)	76	
	30,20	491	(79)	945	(80)	25		37		205	(92)	1655	
	50,30	404	(33)	970	(17)	151	(93)	997	(98)	234	(69)	3825	(86)
	70,50	731	(17)			360	(67)	8744	(89)	310	(41)	1782	(58)
random	10,5	3		2		11		4		2		3	
	15,10	57		40		8		8		30		34	
	20,15	375	(87)	349	(90)	18		26		235	(92)	744	
	30,15	992	(39)	783	(69)	10		15		583	(78)	2356	(99)
	30,20	1601	(39)	735	(43)	103	(98)	117		315	(55)	6878	(90)
	50,30	473	(13)	2210	(7)	325	(80)	1523	(93)	427	(43)	4140	(52)
	70,50	77	(8)			595	(50)	3759	(63)	346	(33)	2057	(43)

Random models are harder to satisfy when the number of tasks and constraints increase, so running algorithms on completely random models without checking a priori whether they are satisfiable or not, share characteristics with satisfiable models for small numbers of tasks and constraints and with the unsatisfiable models for large numbers. We notice that we only see a small bias for our algorithms as the times are not significantly larger and percentages not significantly lower than for the satisfiable and unsatisfiable models. Unfortunately, the best algorithm (7) seems to be the one preferred the most by our testing method, reinforcing that we should do experiments with real-life models.

5 Conclusion and Future Work

We have presented algorithms for translating declarative workflow models specified using Declare to finite automata using LTL for analysis and enactment of the model. The best of our algorithms are several orders of magnitude faster than the previous best algorithm. We have obtained the speed-up by exploiting characteristics of LTL formulae originating from a Declare specification, especially that they are conjunction of simpler formulae defined by the individual constraints of the models. We use automaton operations (automaton product) instead of computing the automaton directly for the conjunction. By structuring the product in balanced trees, grouping automata sharing atomic propositions together, and partitioning the formulae according to the tasks constrained by the originating constraints, we obtain speed-ups of a factor of 1.000-10.000 and more, allowing us to handle models with 50 constraints in seconds, improving on the previous state-of-the art of handling models with 10-15 constraints in minutes or hours. This shows that simple generic algorithms are beat by domain-specific algorithm engineering and heuristics exploiting known structure. We believe our improvement is significant, as it allows us to analyse and enact models of a realistic size, rather than just toy examples. Our next step is to try just that. Our experiments show that we have a slight bias towards the best of our algorithms, and it is interesting to see the improvement incurred by the algorithm on a real-life model.

Only little work has been on improving the very specific kind of automata for Declare. Aside from the generic LTL translation algorithm of [6,7] and improvement using single-event automata as described in [14], work has been done on parallel algorithms for computing binary decision diagrams (which is a specific kind of finite automata, see [1]) using automaton operations. [10] deals with arbitrary propositional formulae and uses the structure of the formula to build a tree of automaton operations (and/or/not) and compute the automaton for the final product using a strategy similar to ours. As they focus on general formulae, they are not free to restructure the computation tree, which yields the greatest gain after taking the step from the standard LTL translation algorithm to automaton operations.

We have already made experiments with improvements of our algorithms. While the ideas have not paid off yet, we believe that improvement can obtained. We have made some experiments with parallelized computation of the products organized in trees. Unfortunately, little is gained as the last computation (computing the root of the tree) dominates the computation. Using a parallel algorithm for product or automaton minimization [16] may improve on this. We would like to experiment with improved algorithms for computing products and representing states, such as using binary decision diagrams [1] or a sharing tree [10]. We have investigated dynamic update of the automaton, so a user can add or remove constraints from the system and interactively obtain an updated automaton for interactive model construction and enactment/analysis or for dynamically reconfigurable models. The idea is that when we add an automaton to/remove an automaton from a tree, we only need update intermediate products on path from the new/removed factor to the root. We can improve on time spent for adding a constraint by adding it next to the old root, introducing a new root. Unfortunately, this cannot compete with the algorithms grouping constraints according to tasks (7/8) as we either void the grouping by multiple additions/removals or make the tree unbalanced by inserting according to the grouping. It would be interesting to investigate a way to rebalance the tree respecting a dynamically computed grouping without having to rebuild it from scratch. It would also be interesting to investigate uses of the non-deterministic automaton for a negated model, as we can compute it very quickly (in milliseconds for models with even hundreds of constraints). One idea is to make an approximation of the automaton for the model for quick analysis and guidance, using the negated automaton to weed out spurious errors.

References

1. Bryant, R.E.: Graph Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
2. Chesani, F., Mello, P., Montali, M., Torroni, P.: Verification of Choreographies During Execution Using the Reactive Event Calculus. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 55–72. Springer, Heidelberg (2009)
3. Declare webpage, `http://declare.sf.net`
4. Etessami, K., Holzmann, G.J.: Optimizing Büchi Automata. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 153–168. Springer, Heidelberg (2000)
5. Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some simplified NP-complete graph problems. Theoretical Computer Science 1, 237–267 (1976)
6. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple On-the-fly Automatic Verification of Linear Temporal Logic. In: Proc. of Protocol Specification, Testing and Verification, pp. 3–18 (1995)
7. Giannakopoulou, D., Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs. In: Proc. of ASE 2001, pp. 412–416. IEEE Computer Society, Los Alamitos (2001)
8. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford University (1971)
9. Kamp, H.W.: Tense Logic and the Theory of Linear Order. PhD thesis, University of California (1968)
10. Kimura, S., Clarke, E.M.: A parallel algorithm for constructing binary decision diagrams. In: Proc. of ICCD 1990, pp. 220–223 (1990)
11. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, Heidelberg (1995)
12. Object Management Group (OMG). Business Process Modeling Notation (BPML). Version 2.0. OMG Avaiable Specification
13. Paige, R., Tarjan, R.E.: Three Partition Refinement Algorithms. SIAM Journal on Computing 16(6), 973–989 (1987)
14. Pei, M., Bonaki, D., van der Aalst, W.M.P.: Enacting Declarative Languages Using LTL: Avoiding Errors and Improving Performance. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 146–161. Springer, Heidelberg (2010)
15. Pei, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-Based Workflow Models: Change Made Easy. In: Chung, S. (ed.) OTM 2007, Part I. LNCS, vol. 4803, pp. 77–94. Springer, Heidelberg (2007)
16. Ravikumar, B., Xiong, X.: A Parallel Algorithm for Minimization of Finite Automata. In: Proc. of IPPS 1996, pp. 187–191. IEEE Computer Society, Los Alamitos (1996)