A Compositional Automata-based Semantics for Property Patterns

Kalou Cabrera Castillos[1], Frédéric Dadeau[1], Jacques Julliand[1], Bilal Kanso[2] and Safouan Taha[2]

[1] FEMTO-ST/DISC - INRIA CASSIS Project
16 route de Gray 25030 Besançon cedex, France

[2] SUPELEC Systems Sciences (E3S) - Computer Science Department
3 rue Joliot-Curie F-91192 Gif-sur-Yvette cedex, France

{kalou.cabrera,frederic.dadeau,jacques.julliand}@femto-st.fr
{bilal.kanso,safouan.taha}@supelec.fr

Abstract. Dwyer et al. define a language to specify dynamic properties based on predefined patterns and scopes. To define a property, the user has to choose a pattern and a scope among a limited number of them. Dwyer et al. define the semantics of these properties by translating each composition of a pattern and a scope into usual temporal logics (LTL, CTL, etc.). First, this translational semantics is not compositional and thus not easily extensible to other patterns/scopes. Second, it is not always faithful to the natural semantics of the informal definitions.
In this paper, we propose a compositional automata-based approach defining the semantics of each pattern and each scope by an automaton. Then, we propose a composition operation in such a way that the property semantics is defined by composing the automata. Hence, the semantics is compositional and easily extensible as we show it by handling many extensions to the Dwyer et al.'s language. We compare our compositional semantics with the Dwyer et al.'s translational semantics by checking whether our automata are equivalent to the Büchi automata of the LTL expressions given by Dwyer et al. In some cases, our semantics reveals a lack of homogeneity within Dwyer et al.'s semantics.

Keywords: Formal Methods, Temporal Properties, Compositional Automata Semantics, Temporal logics, Property Patterns.

1 Motivations

Dynamic properties are commonly described by temporal logics such as the Linear Temporal Logic (LTL). These formalisms are difficult to appropriate by system designers and validation engineers. In order to ease their understanding and writing, Dwyer et al. (denoted DAC in the reminder of the paper) propose in [4, 5] a language of properties based on the composition of predefined patterns

and scopes. A pattern expresses a temporal property on executions seen as sequences of states/events. A scope determines the parts of executions on which the pattern must hold.

In these works, we can measure how much it can be difficult to express properties directly by temporal formulæ. For example, with the language of DAC, one can express the following property: "the state property P' responds to the state property P between the state properties Q and R" by composing the pattern P' responds to P and the scope between Q and R. The corresponding LTL formula given by DAC is: $\Box((Q \wedge \neg R \wedge \Diamond R) \Rightarrow (P \Rightarrow (\neg R \ \mathsf{U} \ (P' \wedge \neg R))) \ \mathsf{U} \ R)$. Even if the specifier is familiar with LTL, these formulæ are very difficult either to write or to understand due to the huge semantics gap between the intuitive formulation of the property in the natural language and its complex and error-prone translation into LTL.

Besides the natural semantics of the patterns and scopes, DAC provide formal semantics by translation into many temporal logics, mapping each pattern/scope combination to a corresponding temporal formula. As there are 10 patterns and 5 scopes, they had to translate the 50 combinations [3]. DAC also noted many possible patterns/scopes variants [3], but they do not support them because translating more than 20 pattern and 20 scope variants requires up to 400 temporal formulæ. Moreover, DAC have defined informally generic patterns (e.g. a first chain of events precedes a second chain of events) that they do not succeed to translate into equivalent generic temporal logic formulæ. Hence, they translated a limited number of obvious cases (e.g. chains having only 1 or 2 events). **Extensibility** and **Genericity** are the main limitations of such a translational semantics.

Furthermore, this translational semantics arises two consistency limitations:

Faithfulness : DAC claim that the temporal formulæ were primarily validated by peer review amongst the project members and then tested against some (un)satisfying sequences of states/events. Hence, we have no formal guarantee that the translated temporal formula is faithful to the intended natural semantics associated to the pattern/scope combination;
Homogeneity : DAC define their language by clearly separating both pattern and scope notions. From a user point of view, a pattern (resp. scope) has a unique natural semantics never mind the scope (resp. pattern) with which it is combined. By adopting translational semantics, they flattened this key separation and translated each pattern and each scope many times into different formulæ corresponding to the different possible combinations. Hence, the same pattern (resp. scope) may have different interpretations according to the scope (resp. pattern) with which it is combined.

In this work, we want a specification language (1) to make easier the expression of the temporal properties by relying on the predefined patterns and scopes of DAC [5]. This language must be easily extensible (2) by adding new variants of patterns and scopes thanks to a compositional semantics. Finally, we intend to adopt an automata-based semantics (3) that is well-adapted to verify

properties, and to generate and evaluate tests because it is provided with many usual structural coverage criteria.

These motivations bring three main contributions that we present in this paper. First, we define a compositional semantics giving an automaton semantics combining the automata of any pattern and any scope. Second, we compare this compositional semantics w.r.t. the LTL translational semantics given by DAC. We will show that even though they focused on translating few specification patterns, they give non-homogeneous interpretations to some patterns and scopes when writing the LTL formulæ. Third, We will give support to many generic patterns and many scope variants emphasizing the extensibility of our compositional semantics.

The paper is structured as follows. Sec. 2 recalls the property language proposed by DAC Sec. 3 presents the compositional semantics of this language by means of automata and their composition. Sec. 4 compares our semantics w.r.t. DAC's semantics and presents the automatization process of our approach using an LTL transformation tool into Büchi automata and a model-checking environment to prove that our automaton is (or is not) equivalent to the LTL formula. Sec. 5 shows the extensibility potential of the language and its semantics. Finally, Sec. 6 concludes and gives some future works.

2 Dwyer et al.'s Property Specification Language

DAC have proposed a pattern-based approach [4]. This approach uses specification patterns that, at a higher abstraction level, capture recurring temporal properties. The main idea is that a temporal property is a combination of one **pattern** and one **scope**. A scope is the part of system execution paths over which a pattern must hold.

Patterns The patterns are temporal conditions on the system executions. DAC propose the ten following patterns classified in the left side of Fig. 1.

- always P: the state property P must hold in all states,
- never P: the state property P does not occur in any state,
- eventually P: the state property P occurs at least once,
- eventually P at most 2 times: the state property P becomes true (after being false in the preceding state) at most 2 times. In other words, switching from $\neg P$ to P occurs at most twice
- P precedes P': a state property P' must always be preceded by a state property P within the execution interval,
- $(\mathsf{P}_1, \mathsf{P}_2)$ precedes P': a state property P' must be preceded by a sequence of states starting by a state property P_1 and leading to a state property P_2,
- P precedes $(\mathsf{P}'_1, \mathsf{P}'_2)$: a sequence of state properties $\mathsf{P}'_1, \mathsf{P}'_2$ must be preceded by a state property P,
- P' responds to P: a state property P must always be followed by a state property P' within the execution interval,
- P' responds to $(\mathsf{P}_1, \mathsf{P}_2)$: a sequence of states starting by a state property P_1 and leading to a state property P_2 must be followed by a state property P',

– $(\mathsf{P}'_1, \mathsf{P}'_2)$ responds to P: a state property P must be followed by a sequence of states $\mathsf{P}'_1, \mathsf{P}'_2$.

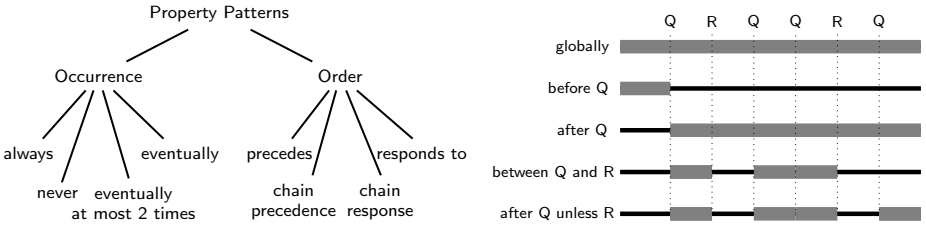


Fig. 1: DAC's Patterns and Scopes

Scopes A scope determines the system execution intervals over which the pattern must hold. In [4], the authors propose five kinds of scopes that are illustrated in the right part of Fig. 1. A property is true if the pattern holds on the execution intervals represented by the thick slices of the sequences. Let p be a pattern and s be a scope, the property p s has the following meaning:

– p globally: the pattern p must hold on the whole execution,
– p before Q: the pattern p must hold before a state property Q occurs,
– p after Q: the pattern p must hold after a state property Q occurs,
– p between Q and R: the pattern p must hold within the system execution intervals from an occurrence of Q to the next occurrence of R,
– p after Q unless R has the same meaning of p between Q and R, but it must hold even if the state property R does not occur.

It is clear that the patterns of DAC dramatically simplify the specification of temporal properties, with a fairly complete coverage. Indeed, they collected hundreds of specifications and they observed that 92% of them fall into this small set of patterns/scopes [4]. Furthermore, DAC adopt translational semantics and provide a complete library [3], mapping each pattern/scope combination to the corresponding formula in many formalisms (e.g. LTL, CTL, Quantified Regular Expressions, μ-calculus). For example, for each scope s, this library maps the property schema P' responds to P s to the equivalent LTL formula as it is given in Tab. 1.

Scope s	LTL
globally	$\Box(P \Rightarrow \Diamond P')$
before Q	$\Diamond Q \Rightarrow (P \Rightarrow (\neg Q \ \mathsf{U} \ (P' \wedge \neg Q))) \ \mathsf{U} \ Q$
after Q	$\Box(Q \Rightarrow \Box(P \Rightarrow \Diamond P'))$
between Q and R	$\Box((Q \wedge \neg R \wedge \Diamond R) \Rightarrow (P \Rightarrow (\neg R \ \mathsf{U} \ (P' \wedge \neg R))) \ \mathsf{U} \ R)$
after Q unless R	$\Box(Q \wedge \neg R \Rightarrow ((P \Rightarrow (\neg R \ \mathsf{U} \ (P' \wedge \neg R))) \ \mathsf{W} \ R)$

Tab. 1: DAC's LTL Mappings of P' responds to P s

We may note that DAC define informally the generic patterns: bounded existence [eventually P at most k times], chain prece-

dence $[(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ precedes $(\mathsf{P}'_1, \ldots, \mathsf{P}'_m)]$ and chain response $[(\mathsf{P}'_1, \ldots, \mathsf{P}'_m)$ responds to $(\mathsf{P}_1, \ldots, \mathsf{P}_n)]$. But because of the translational semantics they can only consider and translate a limited number of cases that are the ten patterns listed above.

3 Compositional Automata-based Semantics

In our approach, the semantics of the temporal properties is defined compositionally by automata composition. Any pattern p is defined by a Büchi automaton pa where the transitions are labeled by state propositions. Any scope s is defined by a specialized Büchi automaton that has a special state, called *composition state* and noted cs, in which a pattern automaton pa can be replaced. Hence, the resulting automaton corresponding to a property p s is defined by substituting the composition state cs of the scope automaton sa by a pattern automaton pa. The resulting automaton is then a Büchi automaton that accepts all the infinite executions (or *runs*) that satisfy the property.

3.1 Pattern and Scope Automata

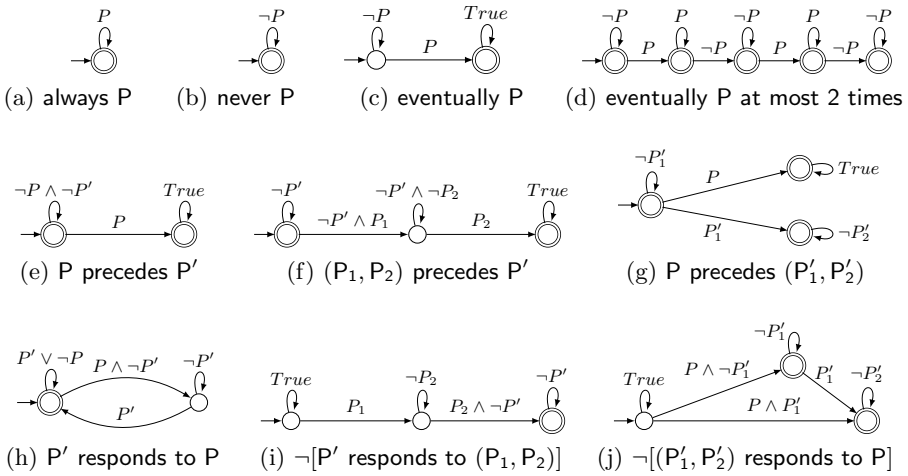
Let $\mathcal{P}$ be a finite set of state propositions. A *Büchi* automaton over $\mathcal{P}$ is a finite-state automaton which accepts infinite words. It is formally defined by a 5-tuple $(Q, init, F, \mathcal{P}, T))$ where Q is a finite set of states, $init(\in Q)$ is the initial state, $F(\subseteq Q)$ is a set of accepting states and $T(\subseteq Q \times \mathcal{P} \times Q)$ is a labeled transition relation.

An infinite word $P_1\ P_2\ \ldots\ P_n\ \ldots$ is accepted by a *Büchi* automaton if there exists a run $q_0 \xrightarrow{P_1} q_1 \xrightarrow{P_2} q_2 \ldots q_{n-1} \xrightarrow{P_n} q_n \ldots$ such that $q_0 = init$ and each step of the run is a transition $(\forall i \in \mathbb{N},\ q_i \xrightarrow{P_{i+1}} q_{i+1} \in T)$ and the set of accepting states within the run is infinite $(\{i \in \mathbb{N} \mid q_i \in F\})$.

While a pattern is described as a Büchi automaton, a scope s is a Büchi automaton which has a composition state cs representing a generic pattern. Hence, a temporal property p s is described by a standard Büchi automaton that is the scope one in which the composition state is substituted by the pattern automaton.

Definition 1 (Pattern and Scope Automata). *Let $\mathcal{P}$ be a finite set of state propositions. A* ***pattern automaton*** *is defined by a Büchi automaton* $\mathsf{pa} \stackrel{def}{=} (Q_{\mathsf{pa}}, init_{\mathsf{pa}}, F_{\mathsf{pa}}, \mathcal{P}, T_{\mathsf{pa}})$ *and a* ***scope automaton*** *by a Büchi automaton* $\mathsf{sa} \stackrel{def}{=} (Q_{\mathsf{sa}} \cup \{\mathsf{cs}\}, init_{\mathsf{sa}}, F_{\mathsf{sa}}, \mathcal{P}, T_{\mathsf{sa}})$ *in which the set of states is the disjoint union of a set of standard states Q_{sa} and a* ***composition state*** *denoted* cs.

Figure 2 illustrates the pattern automata associated to the patterns presented in Sec. 2. The initial states are pointed to by incoming arrows while the accepting states are marked by double circles. We give here the complements of both chain response patterns because the complements are simpler and smaller (3 states instead of 6). The reader may know that Büchi automata are closed under



(a) always P (b) never P (c) eventually P (d) eventually P at most 2 times

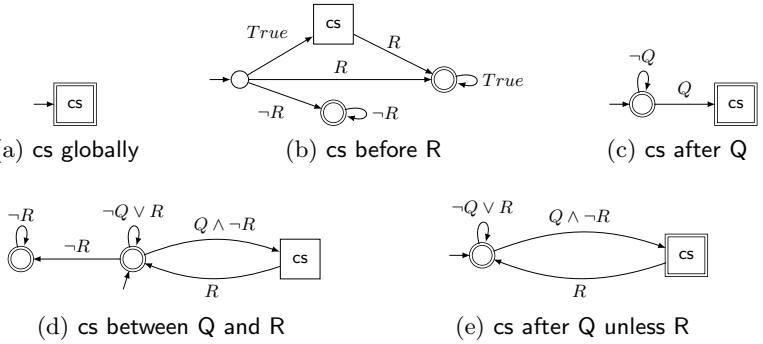
(e) P precedes P′ (f) (P_1, P_2) precedes P′ (g) P precedes (P'_1, P'_2)

(h) P′ responds to P (i) ¬[P′ responds to (P_1, P_2)] (j) ¬[(P'_1, P'_2) responds to P]

Fig. 2: Pattern Automata

complementation and there are many construction algorithms [9]. In Sec. 4, we will explain how we proceed to automatically obtain all these pattern automata.

Fig. 3 illustrates the scope automata associated with the scopes presented in Sec. 2. Squares are used to represent the composition states. Double squares are accepting composition states. In Sec. 4, we will explain how we proceed to automatically obtain all these scope automata.



(a) cs globally (b) cs before R (c) cs after Q

(d) cs between Q and R (e) cs after Q unless R

Fig. 3: Scope Automata

3.2 Composition

In this subsection, we formally define the operation of substitution of the composition state cs by a pattern automaton pa in a scope automaton sa.

Definition 2 (Composition Operation). *Let* $\mathsf{pa} \stackrel{def}{=} (Q_{\mathsf{pa}}, init_{\mathsf{pa}}, F_{\mathsf{pa}}, \mathcal{P}, T_{\mathsf{pa}})$ *be a pattern automaton and* $\mathsf{sa} \stackrel{def}{=} (Q_{\mathsf{sa}} \cup \{\mathsf{cs}\}, init_{\mathsf{sa}}, F_{\mathsf{sa}}, \mathcal{P}, T_{\mathsf{sa}})$ *be a scope au-*

tomaton where cs *is the composition state of* sa*. The* ***substitution of the state*** cs ***by*** pa ***in*** sa *is the Büchi automaton* $(Q, init, F, \mathcal{P}, T)$ *where:*

- $Q = Q_{\mathsf{pa}} \cup Q_{\mathsf{sa}}$
- $init \stackrel{def}{=} \begin{cases} init_{\mathsf{sa}} & \texttt{if } init_{\mathsf{sa}} \neq \mathsf{cs} \\ init_{\mathsf{pa}} & \texttt{otherwise} \end{cases}$ $F \stackrel{def}{=} \begin{cases} F_{\mathsf{sa}} & \texttt{if } \mathsf{cs} \notin F_{\mathsf{sa}} \\ (F_{\mathsf{sa}} \setminus \{\mathsf{cs}\}) \cup F_{\mathsf{pa}} & \texttt{otherwise} \end{cases}$
- $T \subseteq Q \times \mathcal{P} \times Q$ *is the smallest relation defined by the following rules:*

1. *Pattern transitions:* $\dfrac{q \xrightarrow{P} q' \in T_{\mathsf{pa}}}{q \xrightarrow{P \wedge R} q' \in T}$

   where $R \stackrel{def}{=} \bigwedge_{P' \in Out(\mathsf{cs})} \neg P'$ *and* $Out(\mathsf{cs}) = \{P' \mid \exists q''.(q'' \in Q_{\mathsf{sa}} \wedge \mathsf{cs} \xrightarrow{P'} q'' \in T_{\mathsf{sa}})\}$

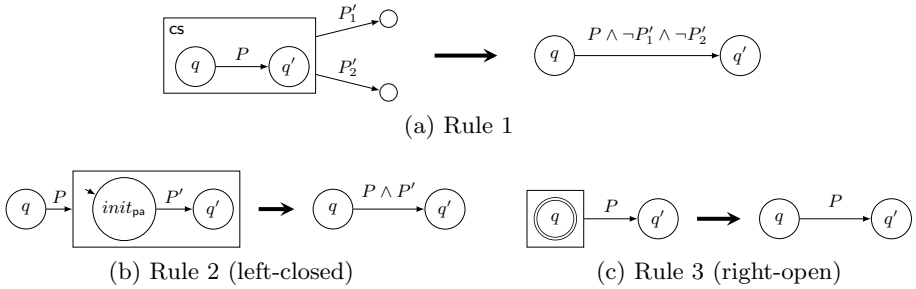
2. *Left-closed scope opening transitions:* $\dfrac{q \xrightarrow{P} \mathsf{cs} \in T_{\mathsf{sa}},\ init_{\mathsf{pa}} \xrightarrow{P'} q' \in T_{\mathsf{pa}}}{q \xrightarrow{P \wedge P'} q' \in T}$

3. *Right-open scope closing transitions:* $\dfrac{\mathsf{cs} \xrightarrow{P} q' \in T_{\mathsf{sa}},\ q \in F_{\mathsf{pa}}}{q \xrightarrow{P} q' \in T}$

4. *Other scope transitions:* $\dfrac{q \xrightarrow{P} q' \in T_{\mathsf{sa}},\ q, q' \in Q_{\mathsf{sa}}}{q \xrightarrow{P} q' \in T}$

The resulting set of states is the union of the sets of states without the composition state cs. The initial state is the initial state of the pattern if the composition state is initial, otherwise it is the initial state of the scope. When the composition state cs is an accepting one, the set of accepting states is the union of both sets of accepting states without cs. Otherwise, it is only composed of those of the scope.

The resulting transitions are defined as follows. The rule 1 adds each transition within the pattern automaton after modifying the label into $P \wedge \neg P'_0 \wedge \cdots \wedge \neg P'_n$ where $P'_i,\ i \in [0, \ldots, n]$ are the labels carried by the n transitions outgoing from the composition state cs as illustrated in Fig. 4(a) where the rectangle represents the composition state cs having two outgoing transitions. This restriction of the labels on the pattern transitions is applied in order to avoid that they capture the scope ones, hence scope transitions keep priority. For example, any transition of the pattern P′ responds to P does not satisfy the condition R which is the exit condition of the scope between Q and R. Indeed, the condition R must not be satisfied (i.e. the exit of the scope must not be possible) before reaching the pattern's accepting state where outgoing transitions hold R (see rule 3). The rule 2 synchronizes the transitions of the scope leading to the composition state cs with the initial transitions of the pattern by making the conjunction of their labels as it is illustrated in Fig. 4(b). For every transition outgoing from the composition state cs, the rule 3 adds a transition from every accepting state of the pattern as illustrated in Fig. 4(c). Rule 2 makes the scope interval left-closed and rule 3 makes it right-open, this aspect will be detailed in Sec. 5. Finally, the rule 4 adds each transition of the scope automaton in which the composition state cs is not involved.



(a) Rule 1

(b) Rule 2 (left-closed)

(c) Rule 3 (right-open)

Fig. 4: Illustration of Composition Rules

Example 1 (Composition of Automata). Fig. 5 shows the Büchi automaton obtained by applying the composition operation given in Def. 2 to the temporal property P′ responds to P between Q and R.

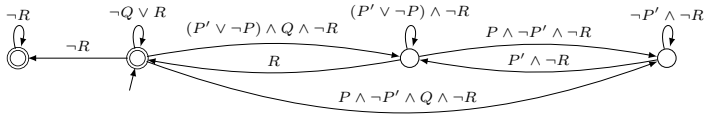


Fig. 5: P′ responds to P between Q and R

Our composition operation is made in a linear complexity w.r.t. the size of the pattern and scope automata. Thus, this automata-based approach yields a technique to transform each DAC temporal property into a Büchi automaton from two Büchi automata in a linear complexity. In contrast, building the same Büchi automaton from the LTL formula given as translation would be exponential w.r.t. the size of the formula [7].

4 Comparison of Both Semantics

In this section, we present the experiments we conducted in order to measure the consistency of our compositional semantics against the translational semantics given by DAC [3]. We do so by comparing our resulting automata with the LTL formulæ given by DAC.

For these experiments, we used the GOAL (Graphical Tool for Omega-Automata and Logics) tool [11] that is an adequate graphical tool for defining and manipulating Büchi automata and temporal logic formulæ. GOAL supports the translation of temporal formulæ such as Quantified Propositional Temporal Logic (QPTL) into Büchi automata where many well-known translation algorithms (e.g. LTL2BA [6]) are implemented and most of them support past operators. It also provides language equivalence between two Büchi automata thanks to efficient complementation, intersection and emptiness algorithms. As the recent implementation of GOAL is based on the Java Plugin Framework, it can be properly extended by new plugins, providing new functionalities that

are loaded at run-time. We implemented our composition algorithm within an independent plug-in that we make available at the web page [10].

The process we applied to do our experiments can be summarized as follows:

1. We used the DAC's LTL formulæ that correspond to the properties combining any pattern with the globally scope to generate the patterns automata using GOAL. These formulæ are shown in Tab. 2 within the globally column. Note that, using our composition operation, the substitution of some pattern p within the scope globally keeps unchanged the automaton pa. This is the way we obtained the pattern automata previously presented in Fig. 2.
2. We used the DAC's LTL formulæ that correspond to the properties combining the always P pattern with any scope to generate the scope automata using GOAL. These formulæ are shown in Tab. 2 within the always P row. Interpreting the unique state having the P loop transition as the composition state, we obtained the scope automata previously presented in Fig. 3.
3. We ran our composition to automatically generate the automata for all pattern/scope combinations. We compared them with the automata obtained directly from the corresponding DAC's LTL formulæ given in [3]. The results of the comparison are given in Tab. 2. For each combination, the automaton of the translational semantics may be equivalent ($\equiv$), strictly included ($\subset$), strictly superior ($\supset$) or not included nor superior ($\neq$) to the automaton given by composition. The non-equivalent cases are indexed by a case number which we use below to explain the reasons behind the mismatching.

	globally	before R	after Q	between Q and R	after Q unless R
always P	$\Box P$	$\Diamond R \Rightarrow (\neg P \ \mathsf{U} \ R)$	$\Box(\neg Q) \vee \Diamond(Q \wedge \Diamond P)$	$\Box(Q \wedge \neg R \Rightarrow (\neg R \ \mathsf{W} \ (P \wedge \neg R)))$	$\Box(Q \wedge \neg R \Rightarrow (\neg R \ \mathsf{U} \ (P \wedge \neg R)))$
never P	$\Box\neg P$	$\equiv$	$\equiv$	$\equiv$	$\equiv$
eventually P	$\Diamond P$	$\subset$ (1)	$\equiv$	$\subset$(2)	$\subset$ (2)
eventually P at most 2 times	$(\neg P \ \mathsf{W} \ (P \ \mathsf{W} \ (\neg P \ \mathsf{W} \ (P \ \mathsf{W} \ \Box\neg P))))$	$\equiv$	$\equiv$	$\equiv$	$\equiv$
P precedes P′	$\neg P' \ \mathsf{W} \ P$	$\equiv$	$\supset$ (3)	$\subset$ (2)	$\subset$ (2)
(P_1, P_2) precedes P′	$\Diamond P' \Rightarrow (\neg P' \ \mathsf{U} \ (P_1 \wedge \neg P' \wedge \bigcirc(\neg P' \ \mathsf{U} \ P_2)))$	$\equiv$	$\equiv$	$\subset$ (2)	$\subset$ (2)
P precedes (P'_1, P'_2)	$(\Diamond(P'_1 \wedge \bigcirc\Diamond P'_2)) \Rightarrow ((\neg P'_1) \ \mathsf{U} \ P))$	$\equiv$	$\equiv$	$\subset$ (2)	$\subset$ (2)
P′ responds to P	$\Box(P \Rightarrow \Diamond P')$	$\equiv$	$\equiv$	$\equiv$	$\equiv$
(P'_1, P'_2) responds to P	$\Box(P \Rightarrow \Diamond(P'_1 \wedge \bigcirc\Diamond P'_2))$	$\supset$ (4)	$\equiv$	$\supset$ (4)	$\supset$ (4)
P′ responds to (P_1, P_2)	$\Box(P_1 \wedge \bigcirc\Diamond P_2 \Rightarrow \bigcirc(\Diamond(P_2 \wedge \Diamond P')))$	$\neq$ (5)	$\equiv$	$\neq$ (5)	$\neq$ (5)

Tab. 2: Comparison between DAC's Semantics and Compositional Semantics

Tab. 3 provides for each mismatching case the formula proposed by DAC and the corresponding formula (verified using GOAL) to the composition automaton we obtained by our composition algorithm. We call this formula *composition formula* and we underline the differences. We use the symbol $\ominus$ for "previous" (resp. B for "back-to") to represent the past-time dual operator of the future operator $\bigcirc$ for "next" (resp., W for "weak-until"). We use the past temporal operators only in case (2) to obtain a concise formula. The reader may know

that past-time modalities do not add expressive power to future linear-time temporal logic but it can be exponentially more succinct [8].

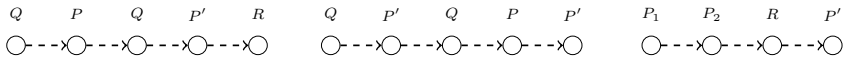
Mismatching cases	**DAC's Formula**		**Composition Formula**
(1) eventually P before R	$\neg R \ \mathsf{W}\ (P \wedge \neg R)$	$\subset$	$\underline{R \vee} \neg R \ \mathsf{W}\ (P \wedge \neg R)$
(2) eventually P/Precedence between Q and R/after Q unless R	$\Box((Q \wedge \neg R) \Rightarrow \dots)$	$\subset$	$\Box((Q \ \underline{\wedge \ominus (\neg Q \ \mathsf{B}\ R)} \wedge \neg R) \Rightarrow \dots)$
(3) P precedes P′ after Q	$\Box\neg Q \vee \Diamond(Q \wedge (\neg P' \ \mathsf{W}\ P))$	$\supset$	$\Box\neg Q \vee (\underline{\neg Q \ \mathsf{U}}\ (Q \wedge (\neg P' \ \mathsf{W}\ P)))$
(4) (P'_1, P'_2) responds to P before R/ . . .	$\dots(P \Rightarrow (\neg R \ \mathsf{U}\ (P'_1 \wedge \neg R \wedge \bigcirc(\neg R \ \mathsf{U}\ P'_2)))) \dots$	$\supset$	$\dots(P \Rightarrow (\neg R \ \mathsf{U}\ (P'_1 \wedge \neg R \wedge \bigcirc(\neg R \ \mathsf{U}\ (P'_2 \underline{\wedge \neg R})))))\dots$
(5) P′ responds to (P_1, P_2) before R/ . . .	$\dots(P_1 \wedge \bigcirc(\neg R \ \mathsf{U}\ P_2) \Rightarrow \bigcirc(\neg R \ \mathsf{U}\ (P_2 \wedge \Diamond P')))\dots$	$\not\supset$	$\dots(P_1 \wedge \bigcirc(\neg R \ \mathsf{U}\ (P_2 \underline{\wedge \neg R})) \Rightarrow \bigcirc(\neg R \ \mathsf{U}\ (P_2 \wedge \underline{(\neg R \ \mathsf{U}\ (P' \wedge \neg R))})))\dots$

Tab. 3: Mismatching Cases

The mismatching case (1) emphasizes that DAC's formula does not recognize the case where R occurs at the initial state, so the interval of the scope is empty since the interval is right-open (see details in Sec. 5) and the property eventually P before R is obviously true. It was an oversight as all other LTL formulæ of the before scope handle such empty interval cases.

Considering case (2), DAC mention in the notes published in [3] that the **first** occurrence of Q opens the intervals of the scopes after Q, between Q and R and after Q unless R (See the right part of Fig. 1). However, some of the proposed formulæ are unfaithful to the **first** occurrence semantics as they consider **all** occurrences of Q. For example, the trace of Fig. 6a that verifies the property P precedes P′ between (first) Q and R, is accepted by our generated composition automaton (equivalent to $\Box((Q \wedge \neg R \ \underline{\wedge \ominus (\neg Q \ \mathsf{B}\ R)} \wedge \Diamond R) \Rightarrow (\neg P \ \mathsf{U}\ (P' \vee R))))$, but it is rejected by the formula given by DAC (i.e. $\Box((Q \wedge \neg R \wedge \Diamond R) \Rightarrow (\neg P \ \mathsf{U}\ (P' \vee R))))$. The past predicate $\underline{\ominus(\neg Q \ \mathsf{B}\ R)}$ (i.e. previous $(\neg Q$ back-to $R)$) ensures that only the first occurrence of Q is considered as there is no occurrence of Q in the past since the last occurrence of R if there is any. We note here that expressing with future modalities the first occurrence of Q following some occurrence of R in between Q and R or after Q unless R, is tedious. For example, the equivalent pure future formula of P precedes P′ between (first) Q and R is:

$$\underline{(\neg(Q \wedge \neg R \wedge \Diamond R) \ \mathsf{W}\ ((Q \wedge \neg R \wedge \Diamond R) \wedge}\ (\neg P' \ \mathsf{W}\ (P \vee R))))$$
$$\underline{\wedge \Box(R \Rightarrow (\neg(Q \wedge \neg R \wedge \Diamond R) \ \mathsf{W}\ ((Q \wedge \neg R \wedge \Diamond R) \wedge}\ (\neg P' \ \mathsf{W}\ (P \vee R))))).$$



(a) case (2) (b) case (3) (c) case (5)

Fig. 6: Mismatching examples

In case (3), the formulæ proposed by DAC consider **any** occurrence of Q $(\Diamond(Q \wedge \dots))$ rather than the **first** occurrence $(\underline{\neg Q \ \mathsf{U}\ (Q \wedge \dots)})$. As a typical example, the trace of Fig. 6b that does not verifiy the property P precedes P′ after Q, is rejected by our generated composition automaton but it is accepted by the formula given by DAC.

Cases (2) and (3) are quite similar and the question "why such a mismatching does not happen for other patterns?" obviously arises. There are two answers depending on cases. In most cases, DAC handle the **first** occurrence semantics within their LTL formulæ, in other cases, patterns are response-oriented properties where the **all** and **first** occurrence of Q semantics are equivalent. For example the LTL formula given by DAC of the property P′ responds to P after Q, i.e. $\Box(Q \Rightarrow \Box(P \Rightarrow \Diamond P'))$ (**all** occurrences) is equivalent to ours $\Diamond Q \Rightarrow (\neg Q \ \mathsf{U} \ (Q \wedge \Box(P \Rightarrow \Diamond P')))$ (**first** occurrence).

DAC have chosen to define scopes as right-open intervals that do not include the state marking the end of the scope [3]. In case (4) and a part of case (5), DAC provide formulæ where P'_2 and R can occur simultaneously; that is unfaithful to the right-open scope semantics.

In case (5), the DAC formula of P′ responds to $(\mathsf{P}_1, \mathsf{P}_2)$ using the modality eventually ($\Diamond R$) does not require that the response P' occurs within the scope! (i.e. before R). For example, the trace of Fig. 6c that does not verify the property P′ responds to $(\mathsf{P}_1, \mathsf{P}_2)$ before R, is rejected by our generated composition automaton but it is accepted by the formula given by DAC.

The experiments, we did here, show the homogeneity of our composition semantics and reveal many different interpretations of the same scope within the translational semantics given by DAC. This emphasizes that it is difficult to give faithful LTL translation to all combinations of patterns and scopes. Our composition semantics brings a valuable consistency.

5 Genericity and Extensibility of the Approach

In the following, we will show the genericity and extensibility of our composition semantics. First we propose some generic patterns and some variant scopes that are not supported by the translational semantics of DAC and then we show their corresponding representation using our automata-based approach.

5.1 Generic Patterns and Variants of Scopes

First, we consider generic patterns that DAC have defined informally but they do not succeed to translate them into equivalent generic temporal logic formulas, hence they only translated a limited number of obvious cases.

- In DAC's work, the pattern eventually has only two forms: eventually P means that P is true at least once and eventually P at most 2 times means that the states switch from ¬P to P at most twice (see Fig. 2). We consider three new generic variants to this pattern: eventually P k times, eventually P at least k times and eventually P at most k times that mean respectively: the state P *becomes true* exactly k times, at least k times and at most k times where k is some natural integer constant.
- Similarly, we propose three generic variants of the eventually pattern considering the number of all occurrences of state P rather than the number of switching occurrences from ¬P to P. We call them precisely P k times, precisely P at least k times and precisely P at most k times.

– Finally, we consider both Chain Precedence and Chain Response patterns having the generic forms $[(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ precedes $(\mathsf{P}'_1, \ldots, \mathsf{P}'_m)]$ and $[(\mathsf{P}'_1, \ldots, \mathsf{P}'_m)$ responds to $(\mathsf{P}_1, \ldots, \mathsf{P}_n)]$.

Next, we propose some enhancements to improve the expressiveness of scopes. These enhancements are inspired by the DAC's notes [3] and our needs within the TASCCC project [1].

– DAC have chosen to define scopes as right-open intervals (i.e. left-closed) that include the state marking the beginning of the scope, but do not include the state marking the end of the scope. We extend scopes with support to open the scope on the left or close it on the right. Hence, we add one variant for both the before _ and after _ scopes and three supplementary variants for the between _ and _ and after _ unless _ scopes. We chose DAC's semantics as the default semantics.
– In DAC's work, between Q and R and after Q unless R scopes are interpreted relatively to the **first** occurrence of Q (see Fig. 1). We keep the first occurrence as default semantics and we add variants to support the **last** occurrence semantics.

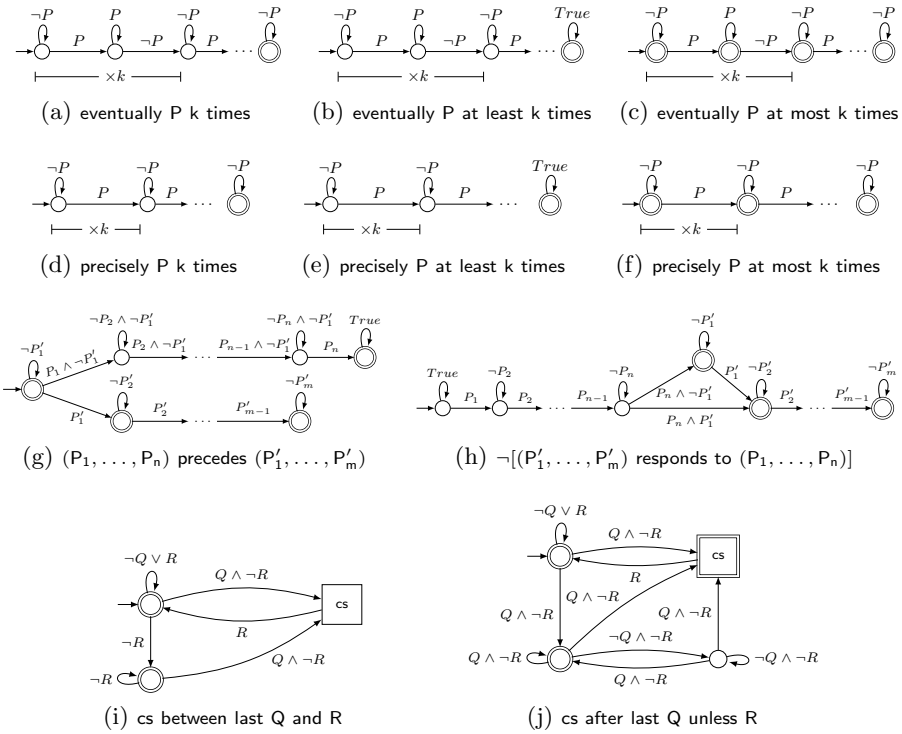
The syntax of our extended pattern-based language is summarized in Fig. 7. Non-terminals are indicated by *italics*, keywords are in policy and terminals are underlined. For example a is an atomic proposition. (...)? designates an optional part. The element P stands for a state property which is a boolean proposition over the alphabet of the different atomic propositions and the optional element '[' or ']' stands for the interval's nature, open or closed at each endpoint.

Property ::= *Pattern Scope*
Pattern ::= always P
| never P
| (eventually | precisely) P ((at least | at most)? integer times)?
| *Chain* precedes *Chain*
| *Chain* responds to *Chain*
Scope ::= globally
| before P ('[' | ']')?
| after ('[' | ']')? P
| between ('[' | ']')? last? P and P ('[' | ']')?
| after ('[' | ']')? last? P unless P ('[' | ']')?
Chain ::= P | P ',' *Chain*
P ::= a | true | $\neg P$ | $P \vee P$

Fig. 7: Syntax of Enriched DAC's Temporal Properties

5.2 Variant Semantics

The generic patterns and the **last** variants of scopes such as between last Q and R are directly expressed in our approach by describing their suitable automata, since their semantics does not have impact on the composition definition (Def. 2) given in Sec. 3. Fig. 8 shows graphically their associated automata.



(a) eventually P k times (b) eventually P at least k times (c) eventually P at most k times

(d) precisely P k times (e) precisely P at least k times (f) precisely P at most k times

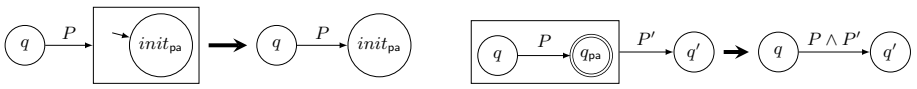
(g) $(P_1, \ldots, P_n)$ precedes $(P'_1, \ldots, P'_m)$ (h) $\neg[(P'_1, \ldots, P'_m)$ responds to $(P_1, \ldots, P_n)]$

(i) cs between last Q and R (j) cs after last Q unless R

Fig. 8: Automata of Generic Pattern and Scope Variants

However, the scopes variants on the closure and opening of the intervals such as before Q] or after] Q require some generalizations in the composition definition. Indeed, in Def. 2, we did not make a distinction between right-open and right-closed intervals, and left-open and left-closed intervals. We have chosen by default the right-open and left-closed intervals as initially given by DAC [4, 3]. An interval left/right-open corresponds to a strict composition while a left/right-closed corresponds to a non-strict composition. A strict composition means that given a composition state cs, its ingoing transitions should be completely executed before the transitions outgoing from the initial state of the pattern automaton are triggered (left-open, see Fig. 9a), and the transitions ingoing in the accepting states of the pattern automaton should be completely performed before the outgoing transitions of cs are triggered (right-open, see Fig. 4c). A non-strict substitution means that the transitions outgoing from the initial state of the pattern automaton should be simultaneously executed with the ingoing transitions of cs (left-closed, see Fig. 4b), and the transitions ingoing in the accepting states of the pattern automaton should be simultaneously executed with the outgoing transitions of cs (right-closed, see Fig. 9b).

Hence, to describe sequencing relationships between the states of the scope automaton and the pattern automaton at the left and the right borders of the composition state, we add the following rules $2'$ and $3'$ to the composition defi-



(a) left-open (b) right-closed

Fig. 9: Illustration of Left-open and Right-closed Composition Rules

nition Def. 2 :

2'. Left-open scope opening transitions: $$\frac{q \xrightarrow{P} \mathsf{cs} \in T_{\mathsf{sa}}}{q \xrightarrow{P} init_{\mathsf{pa}} \in T}$$

3'. Right-closed scope closing transitions: $$\frac{q \xrightarrow{P} q_{\mathsf{pa}} \in T_{\mathsf{pa}},\ q_{\mathsf{pa}} \in F_{\mathsf{pa}},\ \mathsf{cs} \xrightarrow{P'} q' \in T_{\mathsf{sa}}}{q \xrightarrow{P \wedge P'} q' \in T}$$

Due to the compositional semantics we adopted to support the patterns and the scopes proposed by DAC, our language is generic and easily extensible. To add a new variant of any pattern or any scope, it suffices to describe it once in terms of an automaton. This is much easier than specifying all resulting combinations in LTL. We only need to specify the n patterns plus the m scopes to generate the $n \times m$ combinations. In our extension for [1], we have identified above 12 patterns and 21 scopes. To describe them using DAC's semantics, we need to translate 252 combinations whereas following our approach, it suffices to specify the 17 scheme of automata of Fig. 2, Fig. 3 and Fig. 8.

6 Conclusion and Future Work

In this paper, we present a compositional semantics of the DAC's property language. It is defined by the automata of the patterns and the scopes and by the composition operation. We compare it with DAC's translational semantics associating an LTL formula with each pattern/scope combination. This comparison emphasizes the homogeneity of our semantics and reveals that the interpretations of many scopes within their semantics are unfaithful w.r.t the informal definitions given in [4, 5].

Our semantics being compositional, the property language is generic and easily extensible. In this paper, we have shown that handling generic patterns and adding new scope variants, only require to give their semantics by automata. Then, the composition operation gives the semantics of all properties that can be described by combining any new pattern with all existing scopes and combining any new scope with all existing patterns. We also made explicit both the closing and opening default choices of the DAC's semantics by generalizing the composition operation. Moreover, our approach, consisting to choose a scope and a pattern automata, is more efficient for automata-based verification of properties or coverage evaluation of test sequences than a method which consists to choose an LTL formula because it replaces the exponential LTL formula translation into automata by a linear automata composition.

In [2], we are currently using this approach for the evaluation of the coverage of dynamic properties (described as a pattern and a scope composition) by a

test suite. The works that we present here are limited to combine one pattern with one scope. We aim to generalize this work by combining several patterns with a succession of scopes.

References

1. K. Cabrera Castillos, F. Dadeau, J. Julliand, and S. Taha. Projet TASCCC, Test Automatique basé sur des SCénarios et évaluation Critères Communs. `http://lifc.univ-fcomte.fr/TASCCC/`.
2. K. Cabrera Castillos, F. Dadeau, J. Julliand, and S. Taha. Measuring test properties coverage for evaluating UML/OCL model-based tests. In B. Wolff and F. Zaidi, editors, *ICTSS'11, 23-th IFIP Int. Conf. on Testing Software and Systems*, volume 7019 of *LNCS*, pages 32–47, Paris, France, November 2011. Springer.
3. M.B. Dwyer, H. Alavi, G. Avrunin, J. Corbett, L. Dillon, and C. Pasareanu. Specification Patterns. *http://patterns.projects.cis.ksu.edu/*.
4. M.B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, 1999.
5. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In *FMSP*, pages 7–15, 1998.
6. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *(CAV'01)*, volume 2102 of *LNCS*, pages 53–65, Paris, France, july 2001. Springer.
7. 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.
8. N. Markey. Temporal logic with past is exponentially more succinct, concurrency column. *Bulletin of the EATCS*, 79:122–128, 2003.
9. A. P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2-3):217 – 237, 1987.
10. S. Taha. OCL temporal extension, http://wwwdi.supelec.fr/taha/temporalocl/. 2012.
11. Y.K. Tsay et al. Graphical Tool for Omega-Automata and Logics. `http://goal.im.ntu.edu.tw/wiki/doku.php`.