Planning for Declarative Processes

Yoann Laurent, Reda Bendraou, Souheib Baarir, Marie-Pierre Gervais

▶ To cite this version:

Yoann Laurent, Reda Bendraou, Souheib Baarir, Marie-Pierre Gervais. Planning for Declarative Processes. SAC'14 - The 29th Annual ACM Symposium on Applied Computing, Mar 2014, Gyeongju, South Korea. pp.1126-1133, 10.1145/2554850.2554998 . hal-01088183

HAL Id: hal-01088183
https://hal.science/hal-01088183

Submitted on 27 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Planning for Declarative Processes

Yoann Laurent
LIP6
UPMC Paris Universitas
France
yoann.laurent@lip6.fr

Reda Bendraou
LIP6
UPMC Paris Universitas
France
reda.bendraou@lip6.fr

Souheib Baarir
LIP6
University of Paris Ouest
Nanterre
France
souheib.baarir@lip6.fr

Marie-Pierre Gervais
LIP6
University of Paris Ouest
Nanterre
France
marie-pierre.gervais@lip6.fr

ABSTRACT

Recently, declarative process modeling have gained a wide attention from both industry and academia to model loosely-structured processes, mediating between flexibility and support. Instead of describing step by step in an *imperative* way the set of activities to perform (e.g., Petri-net, UML Activity, BPMN), declarative languages define constraints between the process activities that must not be violated during the execution. Even if these languages allow for a high degree of flexibility, this freedom leads to some understandability problems. Indeed, having a mental representation of the possible process executions becomes too complex for humans as the number of constraints increases on the model. This paper presents a novel and formal approach to automatically synthesize execution plans of declarative processes. At design-time, the plans can increase the understanding and the confidence in the model by providing an early and direct experience with it while being modeled. At run-time, the planning component is primordial to ensure that an execution may still lead to a desired goal by giving the possible execution traces leading to it. A working implementation based on the Alloy model-finding method [10] has been developed. The evaluation of this implementation showed us that plans can be generated efficiently and quickly.

Categories and Subject Descriptors

K.6.1 [**Project and People Management**]: Strategic information systems planning; H.4.1 [**Office Automation**]: Workflow management; I.2.4 [**Knowledge Representation Formalisms and Methods**]: Predicate logic, Temporal logic

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'14 March 24-28, 2014, Gyeongju, Korea.
Copyright 2014 ACM 978-1-4503-2469-4/14/03 ...$15.00.

Keywords

Process model, Declarative, Planning, First-order Logic, Alloy

1. INTRODUCTION

Traditional process modeling languages (PML) such as Business Process Modeling and Notation (BPMN), UML Activity Diagram or more formal languages such as Petri-Nets [16] and Pi calculus [14] define a process model as a detailed specification of a step-by-step procedure that should be followed during the process execution. These PMLs have their roots in imperative programming languages, adopting concepts such as conditional branching and loops in order to represent the execution flow in the process model. These approaches strictly specify *how* the process will be executed and yield *highly structured processes*. Their enactment is generally carried-out by a dedicated engine which takes as input the process and automatize its execution.

In recent years, declarative languages have gained increasingly in popularity to model *loosely-structured processes*, balancing between support and flexibility [18]. These languages are based on declaring the different elements of the process (i.e., activities, resource, data and so on) and applying *constraints* on these elements that must hold during the process execution. In this case, the process execution is driven by the constraints: everything that does not violate a constraint is enabled for execution and at the end of the execution all the constraints must be satisfied. Table 1 shows a small sample of control-flow related constraints which can be applied on the activities of the process.

The key difference between declarative and imperative languages for process modeling is that in the former, everything is permitted unless explicitly prohibited by the constraints, while in the second, everything is prohibited unless explicitly specified by the workflow. The major limitations of imperative languages is the fact that most decisions about the execution are already made during the process modeling phase and must be unfolded and modeled explicitly. This leads sometimes to complex process models that are hard to understand and maintain causing some problems with respect to the flexibility of process management systems [22]. Flexibility is the ability to deal with the increasingly wide

Graphical	Predicate	Description
N..M a	`existence[a,N,M]`	Acitivity `a` must be executed between `N` and `M` times.
a b	`co-existence[a,b]`	If one of the activities `a` and `b` is executed, the other one has to be executed too.
a b	`response[a,b]`	Every time activity `a` executes, activity `b` has to be executed after it.
a b	`precedence[a,b]`	Activity `b` can be executed, only if activity `a` has been executed before it.

Table 1: Some example of control-flow related constraints

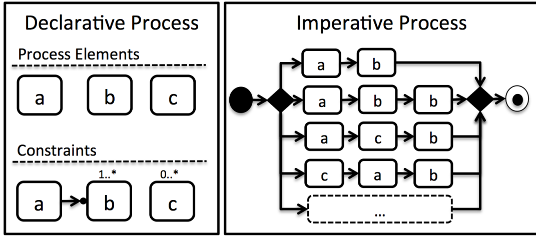

Figure 1: A declarative process with its imperative "equivalence" modeled with UML Activity

range of variations that those systems are subject to in order to remain viable. Declarative languages have been proved to be more suitable for achieving a higher degree of flexibility because they do not require an explicit specification of execution alternatives, but allow for the *implicit* specification of execution alternatives [17]. They support a wide range of flexibility mechanisms such as defer (decide to decide later), change (decide to change the model), deviate (decide to ignore the model), both ad-hoc and evolutionary changes and allow their verification [24]. Figure 1 illustrates the difference between these two paradigms by showing a declarative process with its imperative equivalence. Note that it is not possible to precisely represent this declarative process since the activity "`c`" can be executed at any moment during the process enactment.

1.1 Achilles'heel of Declarative paradigm

Even if declarative process modeling allows for a high degree of flexibility, this freedom comes at the cost of both understandability and maintainability problems [27, 5, 6, 29]. Indeed, declarative process models are hard to read and understand as the number of constraints increases on the model, becoming rapidly too complex for humans to deal with [17]. Especially the *hidden dependencies* [8] which are hard to detect. Figure 1 shows an example of these *hidden dependencies*. Activity "`b`" must be executed during the process enactment due to the `existence[b,1,*]` constraint which requires at least one execution. Activity "`b`" also requires to execute the activity "`a`" before its execution due to the `precedence[a,b]` constraint. Thus, activity "`a`" is obviously executed during the enactment even if this behavior is not *explicitly* visible. The modeler can not only rely on the information displayed explicitly, but also has to carefully examine the process model. These problems are not surprising since cognitive research has demonstrated that imperative programs deliver sequential information much better while declarative programs offer clear insights into circumstantial information [5].

1.2 Planning value for Declarative Processes

Automated planning is related to decision theory and can be a valuable asset to overcome most of the drawbacks of the declarative paradigm. Indeed, planning is a branch of Artificial Intelligence [7] that concerns the production of strategies or plans for a given goal. In the process community, a plan corresponds generally to a sequence of activities to perform which leads to the desired goal, e.g. "which sequence of activities will lead to the availability of `DocumentA` in less than 5 hours?". The sequence of activities to perform and their ordering constitute the output of the planning algorithm. Planning can be particularly helpful during two phases of the process lifecycle [25]:

Modeling. The planning component can help the process modeler to get an *a priori* experience with the model through simulation [21]. This way, the modeler can directly look into some potential execution scenarios to find some flaws inside the design of the model and gain some process understanding. Unlike imperative processes, declarative processes do not suffer from *soundness* problems [26] (e.g. deadlock, livelock and so on) since there is no *tokens-game* driving the execution. However, a declarative model can contain *dead activities* (an activity that can never be executed) and *conflicting constraints* (if there is no execution that would fulfil the set of constraints) [18]. They can also suffer from being *over-constrained* or *under-constrained*. The first case implies that some desired execution scenarios are not available due to the presence of some constraints; the latter implies that some unwanted execution scenarios actually exist due to a lack of constraints. The planning component can cover these verification needs by searching for an execution on which the given activity is executed (*dead activities*), on which the set of constraints is satisfied (*conflicting constraints*) and by ensuring that a plan (not) exists under some circumstances (*over/under-constrained*).

Execution. The planning component can support the execution by (1) proposing plans which align the activities of the process with the availability of the resources and the timing constraints, and (2), proposing plans that will lead to the satisfiability of the desired goals.

1.3 Contributions

This paper proposes an approach that increases the understandability and support of declarative processes by enabling the generation of execution plans for a given goal, while preserving the satisfiability of the process constraints. The main contributions of this paper concern the following points: (1) a first-order logic with relational calculus formalization of the declarative process concepts supporting the control-flow, data-flow, resources, and timing dimensions; including, merging and extending constraints from various sources of the literature [18, 23, 20, 28, 15], (2) a novel approach to automatically generate plans using the Alloy model-finding method [10] and (3) a prototype implementation of the framework, evaluated using case studies from the existing literature.

The key contribution of our approach is not only to help the process modeler in modeling the right trade-off between leeway and constraints, but also to *foreseeing* the future execution in order to avoid process deviations [19].

The paper is organized as follows. Section 2 discusses the related work. Section 3 presents our formalization of the declarative process and planning concepts. Section 4 presents the implementation of our formalization using the Alloy

modeling language. An evaluation of our approach is given in Section 5. Finally, Section 6 concludes by sketching some future perspectives of this work.

2. RELATED WORK

The majority of work around declarative processes are resulting from the valuable work of Van Der Aalst and Pesic on Declare [24, 18]. Declare is a constraint-based system that uses declarative languages for the specification and execution of business processes. One of the advantages of Declare is that its semantics can be characterized in different logic-based approaches, enabling a wide range of reasoning and verification capabilities. The original version of Declare uses Linear Temporal Logic (LTL) over finite traces [20]. Each constraint corresponds to an LTL formula. By building an automaton for each LTL-based constraint it is possible to see whether the execution of an activity will violate the constraint. They also construct an overall automaton based on the conjunction of all constraints to distinguish a constraint that is either in state satisfied, temporarily violated (i.e., the constraint may be satisfied in the future) or violated (i.e., the constraint is not satisfiable in the future). The automaton are also used to drive the process execution by determining the next enabling activities. This way, Declare offers most of the features similar to traditional workflow management systems such as process modeling, design-time verification (*dead activities* and *conflicting constraints*), monitoring, execution and learning from past executions. However, this version of Declare is only defined to tackle control-flow aspects and is not able to check if the model is *over-constrained* or *under-constrained*.

Westergaard et al. [28] propose a timed version of Declare, using MTL (Metric Temporal Logic) [13], a real-time extension of LTL with quantitative temporal operators. They translate the MTL constraints into timed-automata using the UPPAAL model-checker. This way, they extend the original possibility of Declare [18] with the ability to give advices about the actions to undertake in order to obey the latencies and the deadlines specified by the compliance model. The advice corresponds to colored alerts on the model depending on the timing "severity". However, it is up to the user to choose which activities to perform from the advice; no plans are proposed ensuring that it is still possible to perform the process in time.

Montali et al. [15] propose an extension of Declare towards Data-Aware Constraints based on Event Calculus (EC). The key idea is to add *anchor* to activities and then attach some *data* constraints on them. Chesani et al. [2] presents a runtime verification method of web-service choreographies based on a DecSerFlow [23] model (a declarative language tailored towards the specification of service). They select a core set of DecSerFlow elements and formalize them using a reactive version of EC. In these approaches, the trace composed of *event* is checked against the constraints by looking for a given set of *events* on this trace. These approaches allow the users to identify eventual violations only after it has occurred and it is not possible to prevent violations from taking place. EC formalization is primarily thought for monitoring.

Stegan et al. [29] propose to tackle the problems of understandability and maintainability by using the so-called Test Driven Modeling (TDM) methodology. The idea is to define different testcases (a trace execution, the assertion and the terminal condition) and to verify them iteratively on the declarative process. This approach can give some confidence in the model by checking automatically the testcases in a test environment. However, the testcases must include the trace execution and be defined by hand while a planning component can verify the assertion directly on the model without the need of providing a trace.

To summarize, current state-of-art lacks of proposing planning approaches for the declarative paradigm. During the modeling phase, the approaches are confined to the detection of the *dead activities* and the *conflicting constraints* [20]. No approach proposes to ensure that some scenarios are (not) possible on the process. This lack of support concerning the cognitive comprehension of the process may prevent the adoption of the declarative paradigm from the average engineers. During the execution, most of the approaches are only able to monitor the constraints in *a posteriori* way [15, 2], enabling the next possible activities [24, 18] or offering recommendations for decisions based on past experiences [18]. None proposes to generate execution plans increasing the confidence in the model at both design- and run-time.

3. FORMALIZATIONS OF DECLARATIVE PROCESSES

This section presents our first-order logic (FOL) with relational calculus formalization of the declarative process concepts, supporting the control- and data-flow, resources and timing dimensions. Then, this section formalizes the planning problem for the declarative paradigm.

3.1 Declarative Process and Trace

Declarative process consists of *activities*, *resources*, *data* and *constraints*. An activity is a piece of work that is executed by resources. A resource can be an agent, a computer, an equipment or any supply that may be required by an activity. The data are consumed and produced by the activities and can be an integer, a string, an artifact, a document, a model and so on. A constraint specifies a certain rule that should hold in any execution of the model.

DEFINITION 1 (PROCESS). *A* Process *is a tuple* $D = (A, R, D, UseResource)$ *such that:*

- A *is a set of activities,*
- R *is a set of resources,*
- D *is a set of data,*
- UseResource $: A \rightarrow 2^R$ *is the function that maps to each activity a set of resources.*

During the execution of a declarative process, some activities can be executing, some data might be available and the time from the start of the process is continually increasing while performing the activities. Then, we define the notion of state that formalizes the configuration on which the process is at a given time of the process execution:

DEFINITION 2 (STATE). *A state of a* Process $D = (A, R, D, UseResource)$ *is a tuple* $s = (a, d, t) \in (2^A \times 2^D \times \mathbb{N})$ *such that:*

- a *is the set of running activities,*
- d *is the set of available data, and*
- t *is the current discrete time of the execution.*

The set of all states of a process D *is noted* $\mathbb{S} = (2^A \times 2^D \times \mathbb{N})$.

In the following, we use the notation $s.a$, $s.d$ and $s.t$ to access to the corresponding "a", "d" and "t" of a given state s. Since our examples are mostly control-flow based, we only write the running activities instead of the complete tuple to denote a state.

We define the notion of sequence of states as follows:

Definition 3 (Sequence). *A sequence σ of a* Process $D = (A, R, D, UseResource)$ *is a finite ordered set of states denoted $\sigma = \langle s_1, s_2, ..., s_n\rangle \in \mathbb{S}^*$. By notational convenience, we also use σ to denote the set of states of the sequence. The set of all sequences is noted* $\mathsf{Seqs} = \mathbb{S}^*$.

In addition, we define some auxiliary notations and functions to easily manipulate a sequence:

- $|\sigma| = n$ *represents the length of the sequence,*
- *Empty sequence is denoted by* $\langle\rangle$,
- *We use + to concatenate sequences into a new sequence, i.e.* $\langle s_1, s_2, ..., s_n\rangle + \langle s'_1, s'_2, ..., s'_m\rangle = \langle s_1, s_2, ..., s_n, s'_1, s'_2, ..., s'_m\rangle$
- first : Seqs $\rightarrow \mathbb{S}$ *returns the first state of a sequence.* $\mathsf{first}(\langle \mathsf{s_1}, \mathsf{s_2}, ..., \mathsf{s_n}\rangle) = \mathsf{s_1}$.
- last : Seqs $\rightarrow \mathbb{S}$ *returns the last state of a sequence.* $\mathsf{last}(\langle \mathsf{s_1}, \mathsf{s_2}, ..., \mathsf{s_n}\rangle) = \mathsf{s_n}$.
- $\sigma[i]$ *denotes the i-th state of the sequence,*
- next : Seqs $\times \mathbb{S} \rightarrow \mathbb{S}$ *is a bijective function returning the next state of a given state in the sequence:*

$$\mathsf{next}(\sigma, s) \overset{def}{=} \begin{cases} \sigma[i+1] & \text{if } s = \sigma[i] \wedge 1 \leq i < |\sigma| \\ s & \text{if } s = \sigma[|\sigma|] \\ \emptyset & \text{otherwise} \end{cases}$$

- prev : $\sigma \times \mathbb{S} \rightarrow \mathbb{S}$ *is a bijective function returning the previous state of a given state in the sequence such that:*

$$\mathsf{prev}(\sigma, s) \overset{def}{=} \begin{cases} \sigma[i-1] & \text{if } s = \sigma[i] \wedge 1 < i \leq |\sigma| \\ s & \text{if } s = \sigma[1] \\ \emptyset & \text{otherwise} \end{cases}$$

- $\sigma[i, j]$ *denotes all states between $\sigma[i]$ and $\sigma[j]$ inclusively:*

$$\sigma[i, j] \overset{def}{=} (\sigma[i] \cup \mathsf{next}(\sigma, \sigma[i])^*) \cap (\sigma[j] \cup \mathsf{prev}(\sigma, \sigma[j])^*)$$

  Note that $\mathsf{next}(\sigma \in \mathsf{Seqs}, s \in \mathbb{S})*$ *corresponds to the transitive closure of* $\mathsf{next}(\sigma \in \mathsf{Seqs}, s \in \mathbb{S})$.

To represent a process execution, we define the notion of trace: a constrained sequence with global invariants.

Definition 4 (Trace). *A trace of a* Process $D = (A, R, D, UseResource)$ *is a sequence σ_t that respects the following constraints:*

- *(1) The first state of the trace is initialized such that:*

$$\mathsf{first}(\sigma_t).a = \emptyset \wedge \mathsf{first}(\sigma_t).d = \emptyset \wedge \mathsf{first}(\sigma_t).t = 0$$

- *(2) The trace ends properly with no remaining activities executing:*

$$\mathsf{last}(\sigma_t).a = \emptyset$$

- *(3) for each pair of successive states : (i) the time is always equals or increasing and (ii) only one activity can start or terminate:*

$$\forall s \in \sigma_t,\ (\mathsf{next}(\sigma_t, s).t \geq s.t) \wedge (|s.a \ \cup\ \mathsf{next}(\sigma_t, s).a| \in \{|s.a| - 1, |s.a|, |s.a| + 1\})$$

The set of all traces is noted Traces.

For a Process $D = (A, R, D, UseResource)$ and a trace σ_t, we also introduce some auxiliary functions and predicates:

- start/finish : Traces $\times \mathbb{S} \times A$ *is the predicate which determines from a state if a given activity is starting (resp. finishing):*

$$\mathsf{start}(\sigma_t, s, a) \overset{def}{=} a \notin s.a \wedge a \in \mathsf{next}(\sigma_t, s).a$$

$$\mathsf{finish}(\sigma_t, s, a) \overset{def}{=} a \in s.a \wedge a \notin \mathsf{next}(\sigma_t, s).a$$

- activation : Traces $\times A \rightarrow 2^{\mathbb{S}}$ *is the function which returns the set of states on which the given activity is starting:*

$$\mathsf{activation}(\sigma_t, a) \overset{def}{=} \{s \in \sigma_t \mid \mathsf{start}(\sigma_t, s, a)\}$$

To constrain the execution of the process, we define the notion of constraint, i.e. a rule that should be followed during the execution. The constraints are defined on the elements which compose a declarative process and constrain the execution (i.e., the possible traces) on any aspect of the declarative process (control, data, resources and time):

Definition 5 (Constraint). *A constraint is a predicate or a formula which specifies relations between the process elements of a* Process $D = (A, R, D, UseResource)$ *w.r.t a given trace σ_t. We denote* Constraints *the set of all constraints. Let* $c \in$ Constraints. *We denote $\sigma_t \models c$ if σ_t satisfies the constraint c.*

In the following, we give some examples of constraints expressed with the formalism described precedently. Due to space restriction, we choose only some relevant constraints from each aspect of the process dimension (control-flow, dataflow, resources and timing aspect). Interested reader by the full list of constraints we formalized can check the link in Section 5.2.

The `response`(σ_t,`a`,`b`) constraint requires that every time the activity "`a`" terminates, activity "`b`" has to be started after it:

Definition 6 (Response Constraint). response : Traces $\times A \times A$ *is a constraint applied on a trace σ_t, and two activities:*

$$\mathsf{response}(\sigma_t, a, b) \overset{def}{=} \forall s \in \sigma_t,\ (\mathsf{finish}(\sigma_t, s, a) \Rightarrow \exists s' \in \sigma_t\ ,\ s' \in \sigma[\mathsf{next}(\sigma_t, s), \mathsf{last}(\sigma_t)] \wedge \mathsf{start}(\sigma_t, s', b))$$

The `precedence`(σ_t,`a`,`b`) constraint requires that activity "`b`" can be executed only if activity "`a`" has been executed before it:

Definition 7 (Precedence Constraint). precedence : Traces $\times A \times A$ *is a constraint applied on a trace σ_t, and two activities:*

$$\mathsf{precedence}(\sigma_t, a, b) \overset{def}{=} \forall s \in \sigma_t,\ (\mathsf{start}(\sigma_t, s, b) \Rightarrow \exists s' \in \sigma_t\ ,\ s' \in \sigma[\mathsf{first}(\sigma_t), \mathsf{prev}(\sigma_t, s)] \wedge \mathsf{finish}(\sigma_t, s', a))$$

The `existence`(σ_t,`a`,`min`,`max`) constraint requires that activity "`a`" must be executed between "`min`" and "`max`" times:

Definition 8 (Existence Constraint). existence : Traces $\times A \times \mathbb{N} \times \mathbb{N}$ *is a constraint applied on a trace σ_t, an activity and two natural number:*

$$\mathsf{existence}(\sigma_t, a, min, max) \overset{def}{=} |\mathsf{activation}(\sigma_t, a)| \geq min \wedge |\mathsf{activation}(\sigma_t, a)| \leq max$$

The `dataOut`(σ_t,`a`,`d`) constraint requires that at the end of the execution of activity "`a`", the set of data "`d`" must have been created:

Definition 9 (DataOut Constraint). dataOut : Traces $\times A \times 2^D$ *is a constraint applied on a trace σ_t, an activity and a set of data elements:*

$$\mathsf{dataOut}(\sigma_t, a, data) \overset{def}{=} \forall s \in \sigma_t\ ,\ (finish(\sigma_t, s, a) \Rightarrow (next(\sigma_t, s).d = s.d \cup data))$$

The `limitedResourceUse`(σ_t,`r`,`n`) constraint restricts the usage of the resource "`r`" to only "`n`" activity in parallel:

Definition 10 (LimitedResourceUse Constraint). limiteResourceUse : Traces $\times R \times \mathbb{N}$ *is a constraint applied on a trace σ_t, a resource and a natural number such that:*

$$\mathsf{limiteResourceUse}(r, n) \overset{def}{=} \forall s \in \sigma_t\ ,\ (|\{a \in s.a \mid r \in \mathsf{UseResource}(a)\}| \leq n)$$

The `timingExistence`(σ_t,`a`,`from`,`to`) constraint specifies that at least one execution of activity "`a`" must happen in the (discrete) time interval ["`from`","`to`"] of a trace σ_t:

Definition 11 (TimingExistence Constraint). timingExistence : Traces $\times A \times \mathbb{N} \times \mathbb{N}$ *is a constraint applied on an activity and two natural number such that:*

$$\mathsf{timeExistenceStart}(\sigma_t, a, from, to) \overset{def}{=} \exists s \in \sigma_{\mathsf{t}},\ (\mathsf{start}(\sigma_t, s, a) \wedge \mathsf{next}(\sigma_t, s).t \geq from \wedge \mathsf{next}(\sigma_t, s).t \leq to)$$

$$\mathsf{timeExistenceFinish}(\sigma_t, a, from, to) \stackrel{def}{=} \exists s \in \sigma_t, (\mathsf{finish}(\sigma_t, s, a) \land \mathsf{next}(\sigma_t, s).t \geq from \land \mathsf{next}(\sigma_t, s).t \leq to)$$

$$\mathsf{timeExistence}(\sigma_t, a, from, to) \stackrel{def}{=} \mathsf{timeExistenceStart}(\sigma_t, a, from, to) \land \mathsf{timeExistenceFinish}(\sigma_t, a, from, to)$$

At this point, we are able to extend the definition of a process with a set of constraints:

DEFINITION 12 (DECLARATIVEPROCESS). *A* DeclarativeProcess *is a tuple* $CD = (A, R, D, UseResource, C)$ *such that* $(A, R, D, UseResource)$ *forms a* Process *and* C *is the set of* Constraints *of the process.*

To distinguish valid process executions from bad ones, we introduce the notion of valid trace, i.e. a trace on which all the process constraints are satisfied:

DEFINITION 13 (VALID TRACE). *A valid trace of a* DeclarativeProcess $CD = (A, R, D, UseResource, C)$ *is a trace* σ_v *where* $\forall c \in C, \sigma_v \models c$. *The set of all valid traces (i.e. the possible process execution of a declarative process) is denoted* VTraces.

Since our constraints must hold on all traces (universally quantified in def. 13), we will omit them from the parameters list of the constraints that we will use in the remaining sections.

3.2 Planning Problem

The planning problem for a process can be described as follows. Given (1) a process, (2) a current execution (empty or not) and (3) a set of goals (or objectives), which sequence of activities must be performed by the agents to reach the goals? The identified set of sequences of activities to perform is called the set of plans.

This planning problem can be easily described using our formalism. In our case, the process is a declarative process on which any execution are allowed unless explicitly prohibited by the constraints. The current execution can be represented directly as a sequence. In the declarative paradigm, the goals corresponds to a set of *constraints*, e.g. activity "a" and "b" must be executed in less than 5 hours. Thus, under these settings, all valid traces of the process are the so-called plans.

DEFINITION 14 (PLANNING PROBLEM). *Let* $CD = (A, R, D, UseResource, C)$ *be a* DeclarativeProcess. *Let* σ_c *be a sequence corresponding to the current execution sequence. The planning problem is defined by looking for a valid trace* $\sigma_{plan} \in$ VTraces *where* $\sigma_{plan} = \sigma_c + \sigma_{gen}$ *and* $\sigma_{gen} \in$ Seqs. *The set of all solutions of the planning problem (i.e. the valid traces) is denoted* Plans.

EXAMPLE 1. *Consider the declarative process from Figure 1. Let* $CD = (A, R, D, UseResource, C)$ *be this declarative process where* $A = \{a, b, c\}$, $R = \emptyset$, $D = \emptyset$, $UseResource = \emptyset$ *and* $C = \{\mathsf{existence}(\sigma_c, b, 1, \infty), \mathsf{existence}(c, 0, \infty), \mathsf{precedence}(a, b)\}$. *Let* $\sigma_c = \langle\rangle$ *be the current execution sequence. Then, the set* $\{ \sigma_1 = \langle\emptyset, a, b, \emptyset\rangle, \sigma_2 = \langle\emptyset, a, b, c, c, \emptyset\rangle, \sigma_3 = \langle\emptyset, b, b, a, \emptyset\rangle \} \subset$ Traces *and only* $\{\sigma_1, \sigma_2\} \subset$ VTraces *since* σ_3 *violates the constraint* $\mathsf{precedence}(a, b)$.

4. ALLOY FOR DECLARATIVE PROCESSES

Once we formalized declarative processes concepts, traces and planning using FOL, the next step is to choose an implementation language. Alloy [10] was chosen for this purpose. This section gives some background about Alloy and explains how the planning is performed through SAT solving.

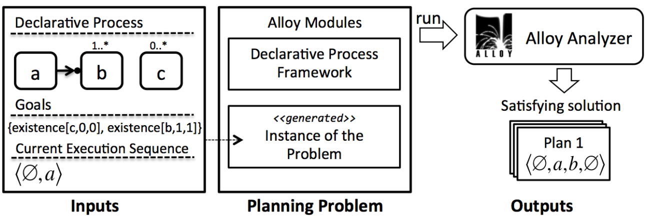

Figure 2: Generation of plans for Declarative Processes using Alloy

4.1 A language and tool for relational models

Alloy is a *declarative* modeling language developed by the MIT and is based on first-order logic and relational calculus for expressing complex structural and behavioral constraints [9]. It is associated to a tool, called `Alloy Analyzer`, a constraint solver that provides fully automatic simulation and checking based on model-finding through SAT-solving [10].

The Alloy language provides a set of concepts allowing to specify elements and constraints using the notions of *signatures*, *relations*, *facts* and *predicates*. A signature (`sig`) defines a set of idioms and relationships between them. An idiom represents an indivisible and immutable entity. The signatures are similar to type declarations in an object-oriented language, and represent the basic entities. Facts (`fact`) are statements that specify constraints about idioms and relationships. These statements must always hold, they are close to the concept of invariants in other specification languages. Predicates (`pred`), as opposed to facts, define constraints which can evaluate to true or false.

Alloy provides two commands to run the `Alloy Analyzer`: `run` and `check`. Command `run` instructs the analyzer to search for an instance satisfying a given formula, and `check` attempts to contradict a formula by searching for a counterexample. Problems given to the `Alloy Analyzer` are solved within a user-specified scope that bounds the size of the domains making it finite and reducible to a boolean formula in order to be checked by the on-the-shelf SAT solver.

4.2 Treating the planning problem

Planning problems are efficiently treated by reducing them to satisfiability (SAT) problems [4, 1]. As described in the precedent section, the `Alloy Analyzer` allows to find a satisfying instance of a problem thanks to the `run` command. Searching for a plan using Alloy is close to the concepts of Constraint Logic Programming over Finite Domains (CLPFD) [12] which allows to declare the conditions that a solution must satisfy and let the solving engine finds variables bindings which leads to an instance. In the case of Alloy, the solving is done through the reduction of an Alloy specification into a SAT problem in a Conjunctive Normal Form (CNF) before presenting it to a SAT solver (MiniSat among others [3]). Then, the SAT solver is able to retrieve a plan by extracting the variables bindings which lead to a solution.

Figure 2 shows the workflow to generate plans for a given declarative process. The planning problem is an Alloy specification composed of two modules. The first one corresponds to the formal framework presented in Section 3, implemented using the Alloy language. The second corresponds to the instance of the planning problem. This instance is generated from the process model, the current execution sequence and the desired goals: actually, we separate the "nominal" con-

```
open declarative //open the framework presented in Section 3
// ---- (1) [Process]
one sig a extends Activity {}
one sig b extends Activity {}
one sig c extends Activity {}
fact constraints {
  atomic[Activity]
  existence[b, 1, integer/max]
  existence[c, 0, integer/max]
  precedence[b, a]
}
// ---- (2) [Goals]
pred goals {
  existence[c,0,0]
  existence[b,1,1]
}
// ---- (3) [Trace]
pred trace {
  setState[State0, none]
  setState[State1, a]
  traces[State1]
}
// ---- [Alloy command]
run {goals and trace}
  for 0 but 3 Activity, 15 State
```

Listing 1: Instance of the planning problem represented using the Alloy framework

straints of the declarative process (we will refer to this set by C_n) from the additional constraints expressed by the users (C_g), called here "goals"[1]. On this example, (1) the process of Figure 1 is used, (2) the goals are defined such as "c" must not be executed and "b" must be executed only one time and (3) the execution sequence is initialized with no running activity on the first state, and the execution of activity "a" on the second state. Listing 1 shows this instance represented using the Alloy framework. As visible on this listing, the transformation to generate this instance is straightforward, i.e. each process element corresponds to a new Alloy `sig` (line 3 to 5); each constraint corresponds to a new call of predicate inside a **`fact`** statement (line 6 to 11); the goals is represented as an Alloy **`pred`** (line 13 to 16); the initial sequence is specified through the call of predicates defined in the framework to constrain the sequence (line 18 to 22). Then, the **`run`** command (line 24) instructs the `Alloy Analyzer` to search for a solution on which the `goal` predicate holds. All the scope of the Alloy signatures (line 25) are straightforwardly determined by the input process model to bounds the size of the domains, e.g. 3 activities on the process imply a scope of 3 for the `Activity` signature. The only exception concerns the scope of the `State` signature, i.e. the maximum length of the generated plan. In this case, we determine the scope of this signature using a simple heuristic based on the existence constraint cardinalities of each activities. Then, we use *incremental-scoping* techniques on the `State` signature to look for bigger plans if required. Then, these modules are given as input to the `Alloy Analyzer` which returns the satisfying instance, i.e. the plans for the given problem.

5. EVALUATION

This section presents our implementation and its evaluation on a case study.

[1]In this context, the process constraints set corresponds to $C = C_n \cup C_g$

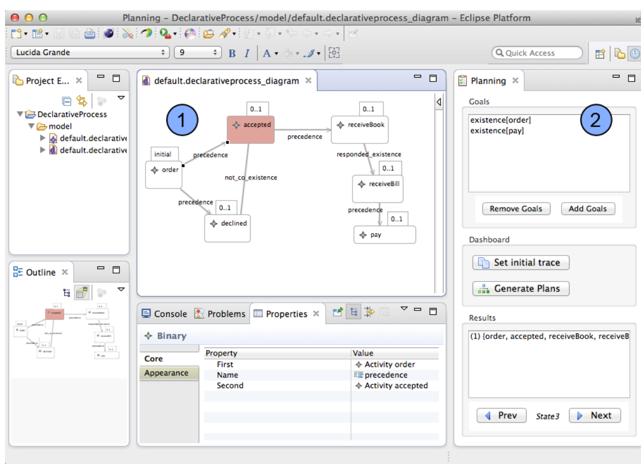

Figure 3: Prototype of the planning component integrated into Eclipse

5.1 Implementation

The prototype we developed is currently provided as an Eclipse EMF plugin. The intent of this prototype is to assist the modeler by automatically generating execution plans for a given declarative process. In order to synthesis the execution plans, the prototype automatically generates the Alloy specification (e.g. see Listing 1) representing the planning problem. Then, the `Alloy Analyzer` is queried using the Alloy API and the result is mapped to the process view to graphically animate the execution of the plan. The modeler never manipulates the Alloy logic. Figure 3 shows a screenshot of the prototype. Label 1 shows graphically the process instance based on the GMF tooling. Label 2 shows the planning component, enabling to set goals (through a predefined template on which the user selects the elements), set the initial sequence (by hand or from a current execution plan), runs the generation, and navigates through an execution plan by animating the process view.

5.2 Case study

To illustrate our approach on a case study, we use the "Acme Travel Company" case from [23]. Acme Travel Service is a fictitious travel agency that has decided to offer its customers the benefit of planning and reserving travel arrangements through a Web based application. Figure 4 shows this process. This process contains 11 activities, 24 constraints and is executed as follows. (1) Acme Travel agency receives an itinerary from the customer (`receive`). (2) After checking the itinerary, the system sends simultaneous requests to the appropriate `airline` and `hotel` agencies to determine which reservations to make. (3) If any of the reservation activities fails (`failed hotel` or `failed airline`), the itinerary is canceled (`compensate`), and the customer is notified of the problem (`notify failure`). (4) Acme Travel waits for confirmations of the reservation requests (`booked hotel` and `booked airline`). (5) When the customer pays the travel (`credit card`), Acme Travel notifies the customer of the successful completion (`notify booked`). It is worth noting that this process allows for many execution alternatives. A fully detailed description of each activity, the constraints, and why the process is modeled that way is available in [23]. However, it is not needed to deeply understand the process to comprehend the need and the use of our approach.

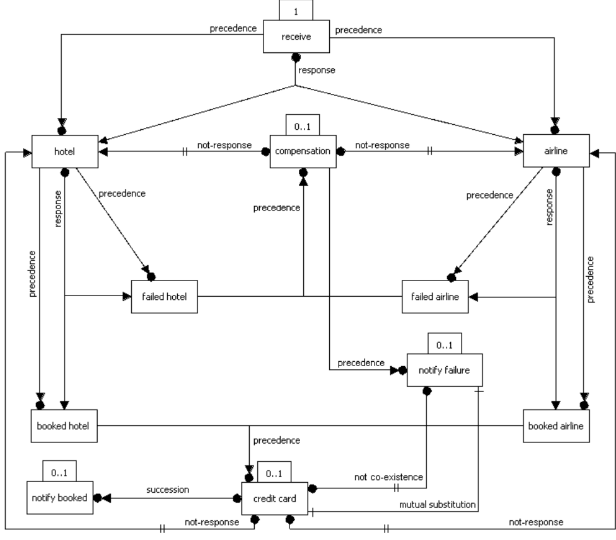

Figure 4: Acme Travel Company case study, taken from [23]

Readers may notice that even if this process is not relatively large, it might be hard to understand and to apprehend the possible process execution without support. One way to gain such understanding can be done through *simulation*, i.e. execution of the process. Then, by starting from an empty sequence $\sigma_{emtpy} = \langle\rangle$ and setting no specific goals such that:

$$C_{g1} = \emptyset \tag{1}$$

the modeler can get an *a priori* experience with the model by looking into the animation of various executions. For instance, the modeler can directly pinpoint that each time, the **`receive`** activity is always executed the first and is followed by either **`hotel`** and **`airline`**. In the case where no plans are found, it means that the process suffers from *conflicting constraints* since no execution is allowed by the constraints of the process. It is also possible to test only a subset of the process constraints by generating only the desired constraint inside the Alloy specification (see Listing 1, line 6 to 11).

The process can be verified against *dead activities* by specifying a goal where the given activity is executed. For instance, to verify that **`notify booked`** is not dead, the goals are specified such that **`notify booked`** is executed at least one time:

$$C_{g2} = \{\mathsf{existence}(\mathsf{notify\ booked}, 1, \infty)\} \tag{2}$$

If no plans are found, it means that **`notify booked`** is dead.

In order to verify that the process is correct, the modeler may want to check that *no* plans exist under some circumstances. For instance, to verify that when the successful completion is sent (**`notify booked`**), the customer can no longer be notified of the failure of the booking (**`notify failure`**) the goals are specified such that:

$$C_{g3} = \{\mathsf{existence}(\mathsf{notify\ booked}, 1, 1), \mathsf{response}(\mathsf{notify\ booked}, \mathsf{notify\ failure})\} \tag{3}$$

Another example might be to ensure that every valid execution of this process, either the **`notify booked`** or **`notify failure`** are executed. Then, the goals are expressed to find an

Current Sequence	Goals	Vars	Clauses	CNF Generation	SAT Solving	Satisfiable?
σ_{emtpy}	C_{g1}	10k	29k	535ms	58ms	yes
σ_{emtpy}	C_{g2}	10k	29k	570ms	54ms	yes
σ_{emtpy}	C_{g3}	10k	29k	580ms	11ms	no
σ_{emtpy}	C_{g4}	10k	29k	597ms	13ms	no
σ_{emtpy}	C_{g5}	10k	29k	550ms	67ms	yes
$\sigma_{runtime}$	C_{g6}	8k	22k	450ms	33ms	yes

Table 2: Metrics for the generation of plans on the Acme Travel case study

execution on which both **`notify booked`** and **`notify failure`** are not executed:

$$C_{g4} = \{\mathsf{existence}(\mathsf{notify\ booked}, 0, 0), \mathsf{existence}(\mathsf{notify\ failure}, 0, 0)\} \tag{4}$$

In both C_{g3} and C_{g4} goals, if the planning component finds an execution, it means that the process is *under-constrained* since the process allows for unwanted executions scenarios.

Another interesting use of the planning component is also to ensure that some plans actually exist. For instance, to ensure that even if the booking of an hotel fail (**`failed hotel`**), it is still possible in the future to get a successful notification of a booking (**`booked hotel`**), the goals are expressed such that:

$$C_{g5} = \{\mathsf{response}(\mathsf{failed\ hotel}, \mathsf{booked\ hotel})\} \tag{5}$$

In this case, if no plan is found, it means that the process is *over-constrained* since a desired execution scenario is not available.

During the execution, the planning component can also give some support to the users. For instance, starting from an execution sequence $\sigma_{runtime} = \{\emptyset, \mathsf{receive}, \mathsf{hotel}, \mathsf{failedhotel}, \mathsf{compensation}\}$ and setting the goals such that **`notify booked`** is reached:

$$C_{g6} = \{\mathsf{existence}(\mathsf{notify\ booked}, 1, 1)\} \tag{6}$$

can ensure that even if the system has the **`compensation`** activity executed, it might still exist an execution on which the booking is successfully done.

Table 2 summarizes the obtained results for the generation of plans based on the Acme Travel case study. Column 1 and 2 represent respectively the current sequence and the goals of the generation. Column 2 and 3 represent the number of clauses and variables of the SAT problem generated from the **`Alloy Analyzer`**. Usually, the complexity of a SAT problem is measured by the number of clauses and variables. Column 4 and 5 represent the time to generate the CNF and to solve the SAT problem. Here, the SAT solving time represents only the generation of one plan. Multiple plans can be generated without the need of re-generating the CNF. Finally, column 6 indicates if the given planning problem is satisfiable, i.e. if at least one plan exists. All analysis were performed on a MacBook Air 2011 with Intel Core i5 processor and 4GB of RAM with Mountain Lion as OS. The results showed us that plans can be generated quickly on a real case from the literature.

It is worth noting that most of the goals defined in this case study used the small set of constraints presented in this paper for the sake of self-containedness. However, our approach allows to define more complex goals such as the given data is available before x time units and so on. Interested readers can download the complete Alloy formalization with more complex examples (with resources, time and data) as well as this case study from our website[2].

[2] `http://pagesperso-systeme.lip6.fr/Yoann.Laurent/`

6. CONCLUSIONS AND FUTURE WORK

This paper proposes an approach enabling the generation of plans for declarative process models. The approach brings to process modelers a valuable aid during both the modeling and execution phases of the process lifecycle by the means of *foreseeing* the future executions.

When compared with the other approaches, our work is the first to use FOL with relational calculus to formalize the declarative concepts, and is more expressive than the traditional LTL-based formalization w.r.t. the data, resources and time dimension. Some constraints are not even expressible in LTL. For instance, it is not possible to specify in LTL that an activity must be executed as many times as another activity since it is not possible to "count" how many times the activity is executed, while our formalization can handle this case easily. Moreover, our formalization is expressible enough to cover declarative constraints from various sources of the literature in a unified way [18, 23, 20, 28, 15].

Currently, the generation corresponds to one/multiple possible plans which satisfy the defined goals. Moolloy is an extension of Alloy implementing a "guided improvement algorithm" (GIA) to solve multi-objective optimization (MOO) problems [11]. Moolloy works by repeatedly adjusting the SAT problem to ask for better and better solutions until no better solution exists, exploring the boundaries of the Pareto Front. We are planning to extend our approach by relying on Moolloy and by adding a set of objectives functions to be optimized during the generation. This way, it will be possible to generate the *optimal* plans for the given goals. For instance, to generate the *fastest* plan on which the given activity is executed. Another interesting use of this extension might be to allow the flexibility by deviation [17], i.e. generating *recovery* plans on which the number of violated constraints are minimized.

Acknowledgments

This work has been funded by the MERgE project (ITEA 2 Call 6 11011).

7. REFERENCES

[1] C. Castellini, E. Giunchiglia, and A. Tacchella. Sat-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147(1):85–117, 2003.

[2] F. Chesani, P. Mello, M. Montali, and P. Torroni. Verification of choreographies during execution using the reactive event calculus. In *Web Services and Formal Methods*, pages 55–72. Springer, 2009.

[3] N. Eén and N. Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

[4] M. D. Ernst, T. D. Millstein, and D. S. Weld. Automatic sat-compilation of planning problems. In *IJCAI*, 1997.

[5] D. Fahland et al. Declarative versus imperative process modeling languages: The issue of understandability. In *Enterprise, Business-Process and Information Systems Modeling*. 2009.

[6] D. Fahland et al. Declarative versus imperative process modeling languages: the issue of maintainability. In *Business Process Management Workshops*, 2010.

[7] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3):189–208, 1972.

[8] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a "cognitive dimensions" framework. *Visual Languages & Computing*, 1996.

[9] D. Jackson. Automating first-order relational logic. In *ACM SIGSOFT Software Engineering Notes*. ACM, 2000.

[10] D. Jackson. *Software Abstractions: logic, language and analysis*. Mit Pr, 2011.

[11] D. Jackson, H.-C. Estler, and D. Rayside. The guided improvement algorithm for exact, general-purpose, many-objective combinatorial optimization. Technical report, MIT-CSAIL-TR-2009-033, MIT Computer Science and Artificial Intelligence Laboratory, 2009.

[12] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM, 1987.

[13] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[14] R. Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[15] M. Montali et al. Towards data-aware constraints in declare. In *SAC*. ACM, 2013.

[16] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[17] M. Pesic. Constraint-based workflow management systems: shifting control to users. 2008.

[18] M. Pesic, H. Schonenberg, and W. M. van der Aalst. Declare: Full support for loosely-structured processes. In *EDOC*, 2007.

[19] M. Pesic, M. Schonenberg, N. Sidorova, and W. M. van der Aalst. Constraint-based workflow models: Change made easy. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*. 2007.

[20] M. Pesic and W. M. van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops*, pages 169–180. Springer, 2006.

[21] A. Rozinat, M. T. Wynn, W. M. van der Aalst, A. H. ter Hofstede, and C. J. Fidge. Workflow simulation for operational decision support. *Data & Knowledge Engineering*, 68(9):834–850, 2009.

[22] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. van der Aalst. Process flexibility: A survey of contemporary approaches. In *Advances in Enterprise Engineering I*, pages 16–30. Springer, 2008.

[23] W. M. Van Der Aalst and M. Pesic. *DecSerFlow: Towards a truly declarative service flow language*. Springer, 2006.

[24] W. M. van Der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science-Research and Development*, 23(2):99–113, 2009.

[25] W. M. Van Der Aalst, A. H. Ter Hofstede, and M. Weske. *Business process management: A survey*. Springer, 2003.

[26] W. M. van der Aalst, K. M. van Hee, A. H. ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011.

[27] B. Weber, H. A. Reijers, S. Zugal, and W. Wild. The declarative approach to business process execution: An empirical test. In *Advanced Information Systems Engineering*, pages 470–485. Springer, 2009.

[28] M. Westergaard and F. M. Maggi. Looking into the future. In *OTM*. 2012.

[29] S. Zugal, J. Pinggera, and B. Weber. The impact of testcases on the maintainability of declarative process models. In *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2011.