Alloy4SPV : A Formal Framework for Software Process Verification

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

Alloy4SPV: a Formal Framework for Software Process Verification

Yoann Laurent[1], Reda Bendraou[1], Souheib Baarir[1,2], and Marie-Pierre Gervais[1,2]

[1] Sorbonne Universites, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France
[2] Universite Paris Ouest Nanterre La Défense, F-92001, Nanterre, France
{yoann.laurent,souheib.baarir,reda.bendraou,marie-pierre.gervais}@lip6.fr

Abstract. In this paper we present a framework for software process verification called Alloy4SPV which uses a subset of UML2 Activity Diagrams as a process modeling language. In order to achieve software process verification, we i) define a formal model of our process modeling language using first-order logic, ii) we give it a formal semantics based on the fUML standard, and iii) we implement this formalization using the Alloy language [1]. In order to ease its adoption by process modelers, our framework comes with a graphical tool and a ready to use and customizable set of software process properties. We categorize these properties into two categories, syntactical and behavioral. We extend the set of behavioral properties we identified from the literature with two new categories that we defined, namely, organizational properties which relate to resource management and planning during process execution and business properties which are project/process specific properties.

1 Introduction

In the current state of practice, process model defects are discovered too late, usually at realization time, after the process has proved to be inefficient or having some behavioral issues such as deadlocks, unreachable activities, inefficient use of resources and timing problems. This could have been avoided with adequate process verification tools that would have formally verified the process model before its deployment in real projects. By process verification we mean determining in advance that the process model exhibits a certain desirable behavior.

In the field of business processes many approaches have been proposed for process verification [2,3,4,5,6,7]. These approaches address mainly the verification of some well-known *behavioral properties* that must be guaranteed by all process's executions. The literature addresses essentially what it is called *soundness properties* [7]. These properties guarantee the absence of deadlocks, unreachable activities, and other anomalies that can be detected without domain knowledge.

Software processes are concerned with additional and critical constraints related to their human-oriented nature. They imply many creative tasks that rely on many factors such as time, human agents and resource management. The success of a software process depends then also on the preservation of many best practices and organizational constraints. We call these constraints *organizational properties* and we consider them as a subcategory of *behavioral properties* since a

state space exploration is required to guarantee their preservation for all possible process's executions. Examples of such properties are to make sure, for instance, that the process or an activity will terminate before a given deadline whatever the execution path, make sure that there will be enough agents to perform the activities of the process, etc.

Another point with current process verification approaches is about the formalism and tools they rely on for performing the verification. Whatever the process modeling language, a formal semantics is given to the language by mapping its constructs to either variants of automata [2,8], Petri Nets [9,10,7,6,5] or process algebra [3,4]. However this means that we are relying on the semantics and concepts of the targeted formal language in terms of expressiveness, e.g. Petri Nets, instead of the modeling language itself. Even if Petri Nets (with their different variants) can represent anything defined in terms of an algorithm, this does not imply that the modeling effort is acceptable. Van der Aalst's paper [11] gives concrete examples of some *Workflow Patterns* that need very complex Petri Nets extensions and tricks to represent them while this is expressed very naturally in UML Activity diagrams (AD) [12].

The approach we promote in this paper through our framework Alloy4SPV is different in the sense that we define the formal semantics of the process modeling language using Alloy instead of relying on the semantics of any of the abovementioned formal languages. Alloy is a declarative modeling language based on first-order logic and relational calculus for expressing complex structural and behavioral constraints [1]. Alloy's logic is quite generic and does not commit to a particular specification style [13]. We believe that this is more natural and allows to preserve the expressiveness of the process modeling language.

As a software process modeling language (SPML), Alloy4SPV uses UML2 Activity Diagrams (AD) which have been given recently a precise execution semantics through the new OMG's fUML standard (Foundational UML) [14]. The choice of UML AD is motivated by the fact that AD are part of a standard widely used in the industry, it has been identified as a good SPML candidate [15,16], a good tooling support is provided, and it supports most of the workflow patterns as identified by [17]. However, it is worth noticing that our approach is applicable to other languages such as the BMPN which is more used in the business process community.

Finally, Alloy4SPV comes with a graphical tool that includes a ready to use and configurable set of process properties in order to ease its adoption by process modelers. Our main goal is to gather under the same umbrella a graphical tool for software process modeling, execution and verification which supports all kinds of properties and most of all which preserves the semantics of the process modeling language. We hope that this would encourage a larger adoption of the process verification discipline and thus, a better management of software projects costs and quality.

The next section starts by introducing the set of properties we identified for software process verification. Section 3 and 4 give the different steps we followed for building Alloy4SPV. An evaluation of our framework is given in Section 5. Related work is given in Section 6. Conclusion and some promising perspectives are sketched in Section 7.

2 Properties for Software Process Verification

In this section we present a categorization of the different properties that can be expressed on software process models. It represents the outcome of a literature review in the business process domain and in software methods and practices.

Over the last decade, many kinds of process properties have been studied [7,5]. They mainly fall into two categories: *syntactical properties* and *behavioral properties*. They are used respectively either to enforce some structural constraints, viewed as invariants, that cannot be expressed with the process modeling language itself or to determine in advance whether a process model exhibits certain (un)desirable behaviors. Even if syntactical errors seem quite obvious to detect by process modelers and enforced by process editors, some constraints may escape the modeler's attention which leads to incorrect process models. This has been confirmed by the study in [18], where 34 process models among 600 of the SAP company process referential were incorrect after analysis.

While the verification of *syntactical properties* is well supported by many approaches [19], they neither guarantee the soundness of process models nor that organizational constraints will be respected. To this aim, in the following we introduce *behavioral properties*. We will give the definition of soundness and focus on the two subcategories of behavioral properties that we introduce, namely *organizational* and *business properties*.

2.1 Behavioral Properties

They express constraints that must be guaranteed by all possible executions of the process. The literature addresses essentially a subcategory of such properties called *soundness properties*. As introduced in [7], soundness tends to check three desirable properties: (i) a started process can always complete (*option to complete*); (ii) it should not have any other activity running when the process ends (*proper completion*); and (iii) the process should not contain activities that will never be executed (*no dead transition*). For instance, to answer the question "will the process terminate?" on the process from Figure 1, a property is expressed to check that whatever the process execution, at the end, the `ActivityFinalNode` is executed. Here, a counter-example is exhibited: $\{Initial, A, Decision, B, Merge\}$, when the `DecisionNode` chooses to execute action `B`, then action `D` can never be executed (since action `C` is not) leading to a deadlock situation.

In the literature, we also find references that relate to soundness focusing on data-flow analysis rather than on control-flow [5]. The goal is to validate the workflow against different data problems such as *missing data*, i.e., when a data element needs to be accessed, but either it has never been created or it has been deleted without having been created again, *inconsistent data*, i.e., if an activity is using this data while another one is writing to it or is destroying it in parallel, and so on.

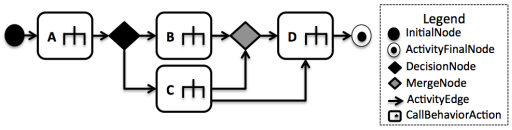
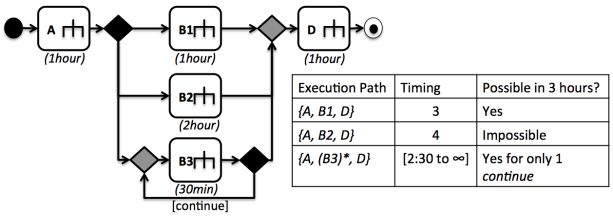


Fig. 1: Example of deadlock in a UML AD due to the control flow



Execution Path	Timing	Possible in 3 hours?
{A, B1, D}	3	Yes
{A, B2, D}	4	Impossible
{A, (B3), D}*	[2:30 to ∞]	Yes for only 1 *continue*

Fig. 2: Example of a UML AD with duration associated to actions

Existing approaches for process verification focus either on control-flow or data-flow, and only few of them ensure both [20]. However, as stated in the introduction, none of them takes into account the particularity of software processes. They treat the process as a simple workflow without covering the range of properties related to the *organizational* or *business constraints* which can highly influence the execution of the process. We introduce these kinds of properties through two new subcategories.

Organizational Properties. They cover organizational constraints about the time to perform the activities of the process, and different kinds of resources (agents, equipment...) problems like *missing resource*, i.e., when an activity requires a resource which may not be available and *inefficient resource use*, i.e., when the resource is inefficiently utilized during the process execution. The goal is to answer questions like: "is it possible to finish the process *on time* whatever the path taken?" "Is the agent always busy?" "Would the process be at activity X before a given deadline?" All these questions are important since they directly influence the decisions taken by the project manager. Figure 2 shows a process on which each activity is associated with a duration and a table displaying the 3 execution paths. Assuming that the process manager plans to do the process in 3 hours, there is 2 cases on which the process will not finish on time: (i) when the `DecisionNode` chooses to execute `B2` and (ii) when the `DecisionNode` chooses to execute `B3` and the decision *continue* is chosen more than once after the execution of `B3`.

Business Properties. While the other categories specify properties that must hold for all processes, business properties represent specific properties tailored to a given process. They play an important role since a process could be syntactically correct and valid against some soundness properties but still violates some business constraints. Therefore, business properties can be used to highlight the importance of a given activity in the process, the fact that one activity should be executed, before, after or between other activities, and so on. Figure 3 shows a simple process considered correct against all the properties from the precedent categories (i.e., syntactical, soundness, organizational). However, `ImportantAction` activity is considered critical in the sense that the process

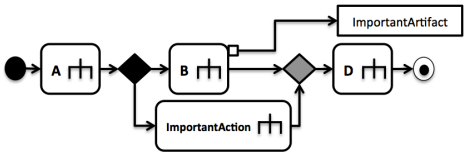


Fig. 3: Example of a correct UML AD

CATEGORY	DEFINITION
(1) Syntactical	
SynWorkflow	Syntactical errors on the workflow of the process *(e.g. the source and target of an edge are different)*
SynOrganizational	Syntactical errors on the organizational part of the process *(e.g. the same agent cannot be assigned more than one time to the same activity)*
(2) Soundness	
OptionToComplete	A started process can always complete
ProperCompletion	No other activity should be running when the process terminates
NoDeadTransition	All the activities must be reachable
Soundness with data	
MissingData	The data are always present when they need to be accessed *(e.g. no data missing to start an activity)*
UselessData	The data created are always used *(e.g. no data created but never used before the process ends)*
InconsistentData	The data can never be in an inconsistent state *(e.g. no data modified by multiple activities in parallel)*
(3) Organizational	
InTime	There is enough time to perform the activities *(e.g. the process will terminate before X hours/days)*
MissingResource	No missing resource to start an activity *(e.g. there are enough agents to do the process)*
InefficientResourceUse	No resources that are inefficiently used *(e.g. the agents have always activity to do)*
(4) Business	
ExistenceActivity	`A` is executed more/less/(between) X (and Y) times
ExistenceTimeActivity	`A` is executed before/after/(between) `X` (and `Y`) time unit
ExistenceTimeData	`ArtefactA` is available before/after/(between) X (and `Y`) time unit
ExistenceTimeResource	`ResourceA` is used before/after/(between) X (and `Y`) time unit
Relation	`A` is executed before/after/in-parallel/in-exclusion/(between) `B` (and `C`)
RelationData	`ArtefactA` is available before/after/in-exclusion of `ArtefactB`
RelationActivityData	`ArtefactA` is available before/after/in-parallel/in-exclusion/(between) the execution of `B` (and `C`)
...	...
LogicBased	e.g. Existence(`A`) `implies` Existence(`B`) `else` Existence(`C`) e.g. Existence(`A`) `implies` (ExistenceData(`ArtefactA`) `and` ExistenceData(`ArtefactB`))
...	...

Table 1: Overview of the software properties we identified

modeler wants it to be executed at least one time during the process enactment. "Is `ImportantAction` executed whatever the choice made during the process execution?" On this example, it is not always the case since the execution path $\{Initial, A, Decision, B, Merge, D, Final\}$ finishes the process without executing `ImportantAction`. Another question here could be: "is `ImportantArtefact` (i.e., the goal of the process) always available at the end of the process? " The other execution path $\{Initial, A, Decision, ImportantAction, Merge, D, Final\}$ shows that it exists a path on which the artefact is not created.

Table 1 summarizes the set of properties we identified. Due to space restrictions we cannot detail all of them. Some of them were already introduced to illustrate the examples while others will be used in Section 5. It is worth noting that the properties distinguish two versions: *weak* and *strong*. The *strong* one ensures that whatever the execution, the property holds while the *weak* one is more permissible and ensures that the property holds for at least one execution. An example of the *weak* and *strong* concepts are given in the case study presented in Section 5.1. Now that we have introduced the set of properties to be integrated into our framework ALLOY4SPV, the next section presents the required steps for the formal software process verification.

3 Formal verification of software processes

The classical approach to achieve the verification of a model (a process model in our case) with respect to a given property, consists beforehand in defining the two entities formally. Then, these entities are submitted to a so-called *model-checker* tool, which will answer to the question of (un)satisfaction of the property by the process model.

In a previous work, we proposed a first-order formalization of fUML for process verification [21]. We have formally reduced the representation of a software process to a vertex-labeled graph. Each graph's node corresponds to a UML Activity node according to its type (i.e. Control, Executable or Object Node). Each graph's arc corresponds to a UML Activity edge (i.e. Control or Object Flow). The execution semantics of this formalism is based on the notions of

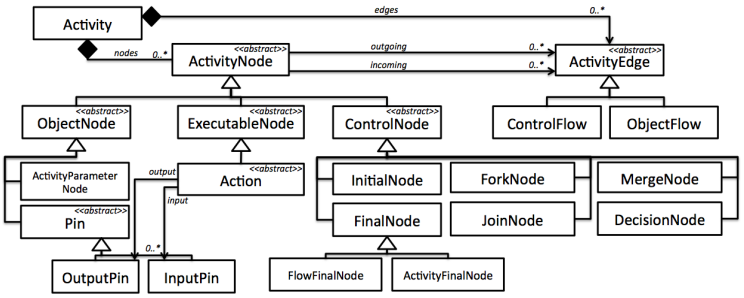


Fig. 4: Excerpt of the fUML Activity meta-model handled by our formalization

states, enabling and *firing* of transitions, similar to those used in the Colored Petri Nets [22]. Figure 4 shows an excerpt of the UML class diagram handled by our formalization. The formalization addresses a subset of fUML encompassing only the concepts required for process modeling as identified in [15]. To be able to reason about each dimension of the process, the formalization covers both *control* and *data-flow* of the process through the use of the AD notations, and takes into account the associated organizational data such as *resources* and *timing* constraints.

3.1 Alloy: a language and tool for relational models

Using our formalization [21], the next step is to choose an implementation language. Alloy [1] was chosen for this purpose. Alloy is a formal language, which has been applied to modelling of systems in a wide range of application domains. It is supported by the `Alloy Analyzer`, a tool, which allows fully automated analysis through SAT solving. Hereunder, we highlight the valuable points that motivated our choice for Alloy:

- It supports a wide variety of properties such as invariants, user-defined assertions, LTL [13] and CTL formulas with fairness constraints [23].
- It is expressive enough to represent a UML-based model associated with OCL constraints [24].
- Alloy's logic is quite generic and does not enforce the user to a particular specification style for modeling and verifying reactive systems.
- It allows one to choose the on-the-shelf SAT-solvers (MiniSat, ZChaff,...).
- It owns a graphical tool as well as an API to integrate seamlessly the verification into a process environment.

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. They 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 counter-example.

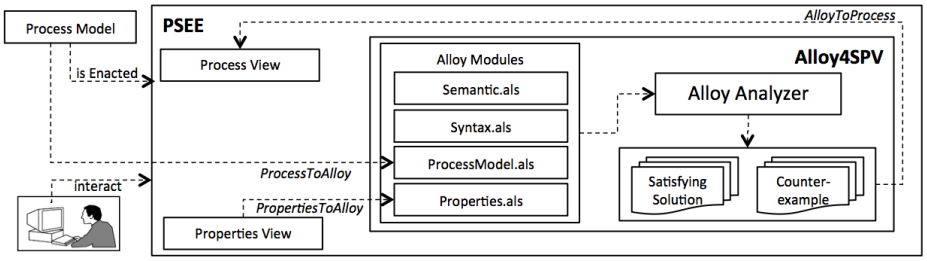


Fig. 5: High-level overview of ALLOY4SPV

4 Alloy4SPV: a Framework for Software Process Verification

This section presents our framework based on the concepts presented so far. We present the high-level overview of our approach and introduce how to represent the different concepts of the software process using the Alloy language in order to enable their automatic verification. Then, we introduce the tool built on top of our framework.

4.1 High-level overview

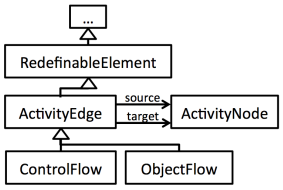
ALLOY4SPV is the name of our framework enabling software process verification. This framework is based on our fUML formalization and is implemented using different Alloy modules. Figure 5 shows an overview of the workflow to achieve the process verification using ALLOY4SPV within a Process-centered Software Engineering Environments (`PSEE`). It takes a `Process Model` in the form of UML2 AD as input. The `Properties View` allows the process modelers to select and express properties through a graphical interface. The `Process View` displays the results about the verification.

ALLOY4SPV is composed of four modules, i.e., `Semantic.als`, `Syntax.als`, `ProcessModel.als` and `Properties.als`. In the following, we detail the content of these static and dynamic ALLOY4SPV modules required by the `Alloy Analyzer` to check a process model. The goal here is to give an overview of the way ALLOY4SPV is implemented using the Alloy language rather than giving an exhaustive definition.

4.2 Static modules

`Syntax.als` represents the syntax of the software process modeling language (SPML). It contains signatures and relations that represent meta-classes and attributes from a subset of the UML AD meta-model (see Figure 4). Listing of Figure 6 shows a sample focusing on the `ActivityEdge`. The signatures follow the hierarchy of the UML AD metamodel.

`Semantic.als` corresponds to the behavioral part of the SPML. It represents the notions of *states, enabling* and *firing* of transitions defined in the formalization. Since Alloy does not commit to a particular specification style, there is no standard way to model and verify reactive systems. However, several patterns have been proposed to address this issue. We adopt the traces pattern [1] to model the sequences of executions of an abstract machine. This pattern imposes




```
abstract sig FumlObject {···}
abstract sig Element extends FumlObject {···}
abstract sig NamedElement extends Element {···}
abstract sig RedefinableElement extends NamedElement {···}
abstract sig ActivityEdge extends RedefinableElement {
  source : ActivityNode,
  target : ActivityNode
}
abstract sig ControlFlow extends ActivityEdge {}
abstract sig ObjectFlow extends ActivityEdge {}
```

Fig. 6: Focus on `ActivityEdge` from `Syntax.als`

a total ordering over the `State` signature and forces that every pair of consecutive states satisfy the given predicate. Listing 1.1 shows a simplified excerpt of this module. The `State` signature represents the configuration on which the process is at a given time of its execution. Therefore, a set of `States` represent a complete execution.

```
open util/ordering[State]
// a State carries the execution information (e.g., tokens, offers, timing and so on).
sig State {
  heldTokens : ActivityNode →one Int,
  offers : ActivityEdge →one Int,
  localClock : ExecutableNode →one Int,
  globalClock : Int,
  running : Status
}
// traces pattern, the regular way to model reactive systems using Alloy
fact traces {
  // constrains all the State to abide from the transition predicate
  all s: State - last | let s' = s.next | {
    s.running = Running implies {
      transition[s,s'] // use ''enabling'' and ''firing'' predicates, defined in the
          formalization
    } else {
      endLoop[s,s']
} } }
```

Listing 1.1: Excerpt of `Semantic.als`

4.3 Dynamic modules

`ProcessModel.als` represents the instance of the process to analyze. Listing of Figure 7 shows a basic process represented using signatures declared in `Syntax.als`. This module is generated from the `Process Model` using a simple model transformation routine, the `ProcessToAlloy` routine, we developed using Java Emitter template (JET); it is basically the Alloy representation of the input `Process Model` [24].

`Properties.als` contains the commands to run the `Alloy Analyzer` over a given set of properties to be checked. Listing 1.2 shows an example of `Properties.als` generated using the `PropertiesToAlloy` routine. The `checkFinal` predicate states that there is some `State` on which the `Final` node is active. Then, the `check` command tries to contradict this predicate by finding a model execution on which there is no `State` with this last property. If the `Alloy Analyzer` finds a counter-example, this means that the `Process Model` is subject to a deadlock. The 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 to be checked by the SAT solver. All the scope of the Alloy signatures are straightforwardly determined by the input


```
// Process - Workflow
one sig Initial extends InitialNode {} {···}
one sig Code extends CallBehaviorAction {} {···}
one sig Final extends ActivityFinalNode {} {···}
one sig InitialToA extends ControlFlow {} {
  source = Initial
  target = Code }
one sig AToFinal extends ControlFlow {} {
  source = Code
  target = Final }
// Process - Organizational
one sig Coder extends Role {}
one sig John extends Agent {}
fun role : Agent →set Role { John →Coder }
fun reqRole : ExecutableNode →set Role { Code →Coder }
fun reqNbAgent : ExecutableNode →Int { Code →1 }
fun reqTiming : ExecutableNode →Int { Code →1 }
```

Fig. 7: `ProcessModel.als` represented in the ALLOY4SPV framework

process model, e.g. 3 `ActivityNode`s on the process imply a scope of 3 for the `ActivityNode` signature. The only exception concerns the scope of the `State` signature, i.e. the trace length on which the process is analyzed, which is determined using *incremental-scoping* technique.

```
pred checkFinal {some s : State | s.getTokens[Final] = 1}
check {checkFinal} for 0 but 5 State, 5 FumlObject, 1 Role, 1 Agent
```

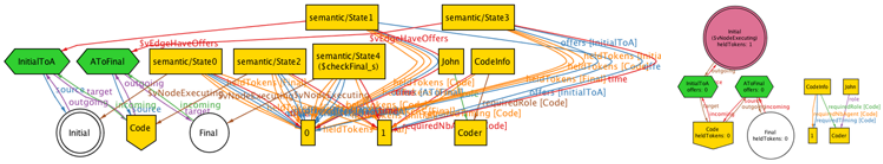
Listing 1.2: Example of verification from `Properties.als`

4.4 Analysis of the results

When satisfying solutions and/or counter-examples are computed by the `Alloy Analyzer`, the results are displayed back to the `Process View` using the `AlloyToProcess` routine. This routine analyzes the results returned by the `Alloy Analyzer` (e.g., extracting the path leading to the deadlock) and displays it on the `Process View`. Figure 8 shows a model found by the `Alloy Analyzer`. In this figure, the model is an instance satisfying the `checkFinal` predicate (`run` command) on the (simple) process from Figure 7. The simple and double stroke circle represent respectively the `ActivityFinalNode` and the `InitialNode`. The hexagons correspond to the `ActivityEdge` while the `ActivityNode` corresponds to the yellow inversed house. Thus, the `AlloyToProcess` routine simply consists in looking through the set of relations of the found model.

4.5 Graphical tool associated to Alloy4SPV

On top of ALLOY4SPV, we have developed a prototype currently provided as an Eclipse EMF plugin. It comes with a library of predefined properties ready



(a) (b)

Fig. 8: (a) Model satisfying the `checkFinal` predicate found by the `Alloy Analyzer`, (b) projected over the first `State` signature

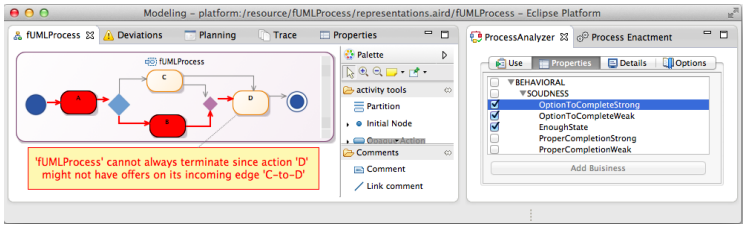


Fig. 9: Process Analyzer using the ALLOY4SPV framework

to be checked and also allows to add some common *business properties* through a graphical interface. The user only has to check in the interface the desired properties, and fill the parameter if required (e.g., maximum time to terminate the process). The *business properties* can be added through pre-defined templates, e.g. select the `ActionA` which must always be executed before `ActionB`. Figure 9 shows a screenshot of our tooling for process modeling and enactment emphasizing the process view and its analyzer. The prototype relies on Obeo UML Designer for modeling and displaying graphically the process. When the verification is performed, the path leading to the counter-example (if any) is highlighted in green for "`run`" properties, and in red for "`check`" properties. Moreover, `CommentNode`s are directly inserted into the model displaying the errors which must be corrected on the model. It is worth noting that the prototype does not require any formal background by the process agent. Everything is automated through the use of the graphical interface to ease tool's adoption.

5 Evaluation

This section presents the evaluation of ALLOY4SPV, by checking some of the properties on a sample of the OpenUP process [25] and on randomly generated processes [26].

5.1 OpenUP case study

We use the software process model illustrated in Figure 10 as a motivating example. It is the `DevelopSolutionIncrement` activity from the OpenUP process [25] represented using UML2 AD. In OpenUP, when a requirement needs to be developed in an iteration, a new `DevelopSolutionIncrement` activity is assigned to a `developer` and a `tester`. The responsability of the `developer` is to create a design and an implementation for that requirement while the `tester` writes and runs developer tests against the implementation to make sure that it works as designed. This activity contains 15 `ActivityNode` and 18 `ActivityEdge`. Note that `ObjectNodes` are excluded for sake of readability.

In the following, some properties from each category of Table 1 are presented. The goal here is to show how the properties are expressed with ALLOY4SPV rather than presenting every single one exhaustively.

(1) Syntactical properties: Testing that each edge has a different source and target is expressed such as:

```
1 pred edgeDifferentTargetSource {
2   all n : ActivityEdge | { not n.source = n.target }
3 }
4 check {edgeDifferentTargetSource} for ···
```

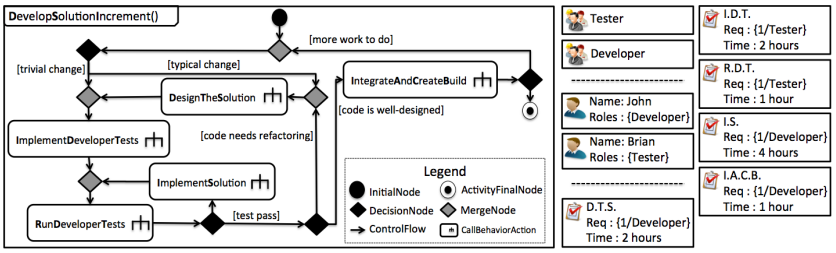


Fig. 10: `DevelopSolutionIncrement` activity from OpenUP

(2) Soundness properties: The *option-to-complete* property is expressed by declaring that at the end, there must be some `State` in which the `Final` node is active:

```
pred OptionToComplete {
  some s : State | s.hasTokens[Final]
}
run {OptionToComplete} for ···
check {OptionToComplete} for ···
```

The `run` command asks the `Alloy Analyzer` to find a model on which the process terminates. If a result is found, it means that there is at least an execution on which the process terminates (*weak option-to-complete*). The `check` command ensures the *strong option-to-complete* by checking for a counter-example on which the execution will not lead to the `Final` node. It is worth noting that the `OptionToComplete` property will always find a counter-example due to the loops inside the workflow. This is because no *fairness* constraints is applied. We eliminate this problem by adding a `fact` constraint inside the `Semantic.als` module that forces *fairness* (i.e., the same outgoing edge cannot be taken infinitely often).

(3) Organizational properties: To check that it is possible to finish the process in less than x hours, the `OptionToComplete` predicate is augmented such that the execution time value at the last `State` is below a given value:

```
pred finalAndTiming[t:Int] {
  OptionToComplete and last.globalClock < t
}
run {finalAndTiming[5]} for ···
```

To check that, at any time during the process execution, there are enough agents to perform the running activities (assuming that all agents are identical) is expressed such as:

```
pred enoughAgent {
  all s:State | #{ node : ExecutableNode | s.hasTokens[node] } < = #Agent
}
check {enoughAgent} for ···
```

In this case, the `enoughAgent` predicate states that at each `State` of the process execution, the number of executing activities is less or equal to the total number of performers.

There is also the possibility to have a finer grained property which takes into account different roles (e.g., coder, designer...) of the agents. The first `check` verifies only for the `Developer` role while the latter verifies all the roles of the process:

Property	Vars	Clauses	CNF Gen.	SAT Solving	Model found?
check edgeDifferentTargetSource	7k	20k	1s	9ms	no
run OptionToComplete	663k	1842k	57s	2s	yes
check OptionToComplete	658k	1840k	56s	15s	no
check enoughAgent	664k	1848k	51s	48s	no
check enoughAgentFor[Developer]	664k	1845k	45s	38s	no
check after[I.S., R.D.T.]	664k	1843k	49s	16s	no
run finalAndTiming[5]	1470k	4612k	125s	23s	yes

Table 2: Metrics from the Alloy Analyser executed on the `DevelopSolutionIncrement` activity

```
pred enoughAgentFor[r : Role] {
  all s:State | #{ node : ExecutableNode | s·hasTokens[node] and r in node·reqRole} <
        = #{ a : Agent | a·role = r}
}
check {enoughAgentFor[Developer]} for ···
check {all r:Role | enoughAgentFor[r]} for ···
```

(4) Business properties: The process modeler may want to check that when the `ImplementSolution` activity is performed, the developed solution is always tested with the `RunDeveloperTests` activity afterward. To express this business property, the process modeler does not have to manipulate the Alloy language but just to select the two actions `ImplementalSolution` and `RunDeveloperTests` and apply the *after* constraint through the Alloy4SPV graphical interface. Thus, the property checks that anytime the `ImplementSolution` is executed, there is some `State` in the future (`s.^next` is the transitive closure of `next` and corresponds to all the following `State` of `s` such as `s.next+s.next.next+s.next...`) on which `RunDeveloperTests` is executed:

```
pred after[a,b:ExecutableNode] { // defined in the Semantics·als module
  all s:State | s·hasTokens[a] implies
    some ss : s·^next | ss·hasTokens[b]
}
check {after[ImplementSolution, RunDeveloperTests]} for ···
```

In order to perform the verification of the aforementioned properties with respect to the part of the OpenUP process in Figure 10, `Alloy Analyzer` reduces the verification to a SAT problem. It is presented to a SAT solver (MiniSat among others) in a Conjunctive Normal Form (CNF) format. A CNF is a conjunction of clauses. Each clause represents a disjunction of variables. A satisfying assignement to a SAT problem consists of a boolean affectation to the variables such that all clauses are satisfied. Usually, the complexity of a SAT problem is measured by the numbers of clauses and variables.

All analyses were performed on a MacBook Air 2011 with Intel Core i5 processor and 4GB of RAM with Mavericks as OS. Table 2 summarizes the obtained results where column 1 represents the analyzed property. Columns 2 and 3 represent, respectively, the number of generated variables and clauses. Columns 4 and 5 represent, respectively, the time to generate the CNF and to solve the SAT problem. Finally, column 6 indicates if a model is found (i.e. satisfiability for a `run` command, and counter-example for a `check` command).

Besides, these results highlight the effectiveness of our tool w.r.t. a concrete example [25]. Actually, even if the whole generated SAT problems present a relatively high complexity (almost 2 million clauses and over 600 thousand variables), the solving time is less than one minute for untimed properties. The

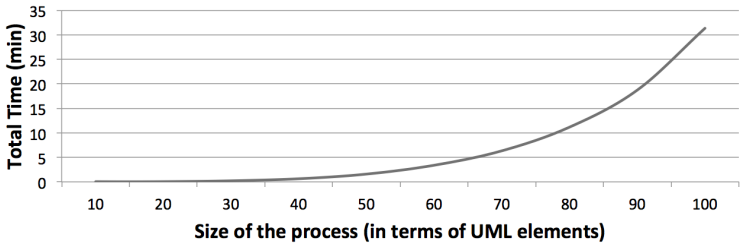


Fig. 11: Total time to check the `OptionToComplete` property depending on the process size

timed-related properties (`run finalAndTiming[5]`) have a similar ratio in terms of clauses and variables but require more time due to the presence of extra states introduced by the clocks to handle the time elapsing. The full details with examples of the ALLOY4SPV modules can be found on our website[3].

5.2 Randomly generated processes

One of the challenges we face to validate our approach is the inability to find realistic data and models. The small set of samples and "toy" models publically available in the literature is insufficient to conduct a serious empirical study to validate works around software process analysis and verification. Moreover, due to privacy reasons, partner companies are reluctant to share their models representing the result of years of best practices and the capitalization of developers and project managers know-how which took time to design.

This problem led us to develop our own process generator [26]. We used it to randomly generate processes ranging from 10 to 100 UML elements. These processes have only control-flow nodes without loops and contain sequential routing (`ControlFlow` edges), action to perform (`OpaqueAction`), parallel routing (`ForkNode`), synchronizer (`JoinNode`), conditional routing (`DecisionNode`) and merging structure (`MergeNode`). Even if our processes are artificial, they present a high-level of realism. The model generator reproduces how a modeler could have developed a process in a real situation. It generates the process through a sequence of Change Patterns [27]. Each process is then checked w.r.t. the `OptionToComplete` property and only models without counter-examples are kept (only the largest verification time is of interest).

Figure 11 shows the solving-times to check this property. These results show that the solving times are reasonable w.r.t. the complexity of the generated models (in terms of number of UML elements). Actually, the generated SAT problem of the model with 100 UML elements contains almost 18 billion clauses and 8 million variables and is resolved in 31 minutes which highlights the fact that our SAT problems belong to a relatively easy-to-solve SAT category. Once again, this emphasizes the effectiveness of our approach.

6 Related Work

There is an extensive literature on verifying process models. Since a lot of the work has been done in the business process community, we do not restrict ourselves to the verification of *software* processes. Generally, the verification is based

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

on mapping the process model into mathematical formalisms used to model systems such as automata, Petri Nets or process algebra.

Many approaches have origins in the Petri Nets formalism, either because the modeling language is based on it (e.g., Workflow Nets [9]) or through a mapping to it [10]. In [9], Van der Aalst et al. introduce the Workflow Nets, a particular class of Petri Nets dedicated to the modeling of workflow with an augmented graphical notation (e.g., AND-splits, AND-joins and so on). In [6] a large number of industrial business processes have been successfully checked on the soundness properties using the LoLa model checker. In [10], the process modeled in UML AD is mapped to Colored Petri Nets [22] in order to enable automatic verification. Due to the fact that Petri Nets enjoy an easily understandable and graphical notation as well as a plethora of mature tools enabling efficient analysis, they have been widely applied in the process analysis field. However, even if the verification of Petri Nets based process is efficient to check properties such as reachability, liveness and boundness, they fail when the system needs to handle a wide variety of data. The use of data on the system multiplies the number of places and introduces some state space explosion problems making the analysis difficult (sometimes impossible). Moreover, these approaches focus only on the soundness properties.

Other approaches use process algebra [3,4], a strict and well-established theory that support the automatic verification of properties of systems behavior as well as Petri Nets. In [3], the authors show how the Communicating Sequential Processes (CSP) algebra can be applied to model complex workflow systems. They use the FDR (Failures-Divergences Refinement) model-checker to automatically check behavioural properties. Liu et al. [4] transforms models expressed in Business Process Execution Language (BPEL) into π-calculus. They also capture compliance rules in the graphical Business Property Specification Language (BPSS) and automatically translate them into temporal logic. This approach is able to handle the verification of both *soundness* and *business properties*. However, process algebra such as π-calculus is limited in the ability to support most of the workflow patterns [17] used in processes.

Further approaches are based on domain-specific language. Eshuis et al. [2] check UML AD in the context of workflow modeling by translating the activity into the input language of NuSMV, a symbolic model checker. The work was done before the finalisation of the UML 2.0 specification, thus the semantics used remains unclear and many assumptions have been made about it. Guelfi et al. [8] propose a translation of UML AD into Promela (Process or Protocol Meta Language) in order to check behavioral properties with the model-checker SPIN. However, no implementation is provided and the set of properties which may be checked are not precise.

In the case of UML AD verification, all these formalisms have been investigated: (1) process algebra using π-calculus [28] and CSP (Communicating Sequential Processes) [29], (2) automaton using NuSMV formalism [2] and Promela (Process or Protocol Meta Language) [8], and (3) Petri Nets formalism through transformation [10]. However, only the work of Abdelhalim et al. [29] is based on the fUML semantics, but lacks by focusing the verification only on deadlocks.

To sum up, most of the approaches focus on verifying control-flow related properties and only a few treat the data on the process. Despite the numerous approaches to check behavioral properties on a process, none of them proposes to

check the *organizational properties*. To our knowledge, no approach proposes to check syntactical and all the behavioral properties in a unified way as promoted by ALLOY4SPV.

7 Conclusion and Future Work

While verification is a critical and an important endeavor in software development, it still remains the Achilles heel of software processes and a main source of their low adoption. Indeed, with the increasing complexity and size of processes, process modelers need adequate tooling support to simulate and to verify their processes before their use in real projects. Some critical processes may reach more than 250 activities, with very complex workflows, dependencies, loops, synchronizations, and without an automated and exhaustive verification, possible sources of inconsistencies and problems may persist. The formalization on which ALLOY4SPV is based is able to deal with control- and data-flow, resources, and timing aspects of the process in a unified way. Therefore, ALLOY4SPV and its associated interface is able to verify automatically a wide range of properties without the user's intervention and allows one to verify some *business properties*. Currently, the tool is under evaluation within the European MERgE project, whose main goal is to develop and demonstrate innovative concepts and design tools addressing both "safety" and "security" concerns in development processes.

The case study and the tool proved the feasibility of our approach, however some improvements to our approach are already under realization. Even if our evaluation shows relatively good performance, we believe that there is still room for improvement. Many optimization techniques can be explored: (1) using *slicing technique*, i.e. partially generates the `Semantic.als` to cope only with the need of the properties; (2) using graph reduction techniques to reduce the size of the process [30]; and (3) treat the properties related to time in a more efficient way based on the expertise of well-known approaches such as timed automata [31].

Acknowledgments. The authors' work is funded by the MERgE project (ITEA 2 Call 6 11011).

References

1. Jackson, D.: Software Abstractions: logic, language and analysis. Mit Pr (2011)
2. Eshuis, R.: Symbolic model checking of uml activity diagrams. TOSEM **15**(1) (2006) 1–38
3. Wong, P., Gibbons, J.: A process-algebraic approach to workflow specification and refinement. In: Software Composition, Springer (2007) 51–65
4. Liu, Y., Muller, S., Xu, K.: A static compliance-checking framework for business process models. IBM Systems Journal **46**(2) (2007) 335–361
5. Trčka, N., van der Aalst, W., Sidorova, N.: Data-flow anti-patterns: Discovering data-flow errors in workflows. In: AISE, Springer (2009) 425–439
6. Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous soundness checking of industrial business process models. Business Process Management (2009) 278–293
7. van der Aalst, W., Van Hee, K., ter Hofstede, A., Sidorova, N., Verbeek, H., Voorhoeve, M., Wynn, M.: Soundness of workflow nets: classification, decidability, and analysis. Formal Aspects of Computing **23**(3) (2011) 333–363

8. Guelfi, N., Mammar, A.: A formal semantics of timed activity diagrams and its promela translation. In: Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific, IEEE (2005) 8–pp
9. van der Aalst, W.M.: The application of petri nets to workflow management. Journal of circuits, systems, and computers **8**(01) (1998) 21–66
10. Jung, H.T., Joo, S.H.: Transformation of an activity model into a colored petri net model. In: TISC, IEEE (2010) 32–37
11. Ter Hofstede, A.: Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In: of DAIMI, University of Aarhus, Citeseer (2002)
12. Wohed, P., van der Aalst, W.M., Dumas, M., ter Hofstede, A.H., Russell, N.: Pattern-based analysis of the control-flow perspective of uml activity diagrams. In: Conceptual Modeling–ER 2005. Springer (2005) 63–78
13. Cunha, A.: Bounded model checking of temporal formulas with alloy. arXiv preprint arXiv:1207.2746 (2012)
14. OMG: Semantics of a foundational subset for executable uml models (fuml) version 1.0. http://www.omg.org/spec/FUML/ (2011)
15. Bendraou, R., Gervais, M.P., Blanc, X.: Uml4spm: A uml2. 0-based metamodel for software process modelling. MoDELS (2005) 17–38
16. Bendraou, R., Jézéquel, J., Gervais, M., Blanc, X.: A comparison of six uml-based languages for software process modeling. Software Engineering, IEEE Transactions on **36**(5) (2010) 662–675
17. van Der Aalst, W.M., Ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and parallel databases **14**(1) (2003) 5–51
18. Mendling, J., Moser, M., Neumann, G., Verbeek, H., van Dongen, B., van der Aalst, W.: Faulty epcs in the sap reference model. Business Process Management (2006) 451–457
19. Hsueh, N., Shen, W., Yang, Z., Yang, D.: Applying uml and software simulation for process definition, verification, and validation. Information and Software Technology **50**(9) (2008) 897–911
20. Trcka, N., van der Aalst, W., Sidorova, N.: Analyzing control-flow and data-flow in workflow processes in a unified way. Computer Science Report (08-31) (2008)
21. Laurent, Y., Bendraou, R., Baarir, S., Gervais, M.P.: Formalization of fUML: an Application to Process Verification. In: CAISE. (2014)
22. Jensen, K.: Coloured petri nets. Petri nets: central models and their properties (1987) 248–299
23. Vakili, A., Day, N.: Temporal logic model checking in alloy. Abstract State Machines, Alloy, B, VDM, and Z (2012) 150–163
24. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: Uml2alloy: A challenging model transformation. Model Driven Engineering Languages and Systems (2007) 436–450
25. Eclipse: Openup. http://epf.eclipse.org/wikis/openup/
26. Laurent, Y., Bendraou, R., Gervais, M.P.: Generation of Process using Multi-Objective Genetic Algorithm. In: Proceedings of the 2013 International Conference on Software and Systems Process, ACM (2013) to be published
27. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features–enhancing flexibility in process-aware information systems. Data & knowledge engineering **66**(3) (2008) 438–466
28. Dong, Y., ShenSheng, Z.: Using π-calculus to formalize uml activity diagram for business process modeling. In: ECBS, IEEE (2003) 47–54
29. Abdelhalim, I., Sharp, J., Schneider, S., Treharne, H.: Formal verification of tokeneer behaviours modelled in fuml using csp. In: Formal Methods and Software Engineering. Springer (2010) 371–387
30. Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Information systems **25**(2) (2000) 117–134
31. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical computer science **126**(2) (1994) 183–235