Verifying Modelling Languages using Lightning: a Case Study

Loïc Gammaitoni, Pierre Kelsen, and Fabien Mathey

University of Luxembourg

Abstract. The formal language Alloy was developed to provide fully automatic analysis of software designs. By providing immediate feedback to users it allows early detection of design errors. The main goal of the Lightning tool is to apply the power of Alloy's automatic analysis to the domain of software language engineering. The tool allows to represent abstract syntax, concrete syntax and semantics of a modelling language in Alloy. In this paper we describe the verification capabilities of Lightning with the help of a concrete modelling language, namely the language of structured business processes.

1 Introduction

The formal language Alloy was developed to "capture the essence of software abstractions simply and succinctly, with an analysis that is fully automatic, and can expose the subtlest of flaws" [9]. By allowing continuous automatic analysis during the design process software modellers can uncover design errors quickly. This design process, which could aptly be called "agile modelling", also stimulates the modellers since it provides immediate feedback.

The goal of the Lightning tool[1] is to apply the power of the Alloy language and its tool, the Alloy Analyzer, to the domain of software language engineering. It was already shown earlier [12] that Alloy is a suitable language for defining syntax and semantics of modelling languages . The Lightning tool can be viewed as a first practical validation of ideas presented in that work.

One can consider Lightning as an important step towards a language workbench based on Alloy. The emphasis of the tool is currently on automatic validation of language definitions using Alloy's SAT-based analysis. All basic components of a modelling language can be defined in the tool: abstract syntax, concrete syntax and semantics. Concrete syntax is currently restricted to visualising language models. Semantics can be specified in the style of operational semantics and its execution can be visualised as well. All specifications of language components and accompanying transformations are defined in Alloy. The tool is, however, not limited to language specifications expressed in Alloy since it allows importing metamodels expressed in Ecore (feature not described in the present paper). For Lightning to become a full-fledged language workbench [5], more sophisticated editor support has to be provided (only exists in rudimentary form in the present version) as well as code generation facilities to interface with existing programming languages.

The main purpose of this paper is a description of the verification capabilities of the Lightning tool. We will examine how the tool assists the user in writing correct language specifications.

This paper is organised as follows: we first describe the case study we will use in this paper. In section 3 we introduce the Lightning tool. We then describe how Lightning assists the user in designing the abstract syntax (section 4), concrete syntax (section 5), and semantics (section 6). We wrap up the paper with a discussion of our contribution in the context of related work and present concluding remarks and future work in the final section.

2 Case Study

In this paper, we illustrate Lightning's verification features by designing a Structured Business Process (SBP) language.

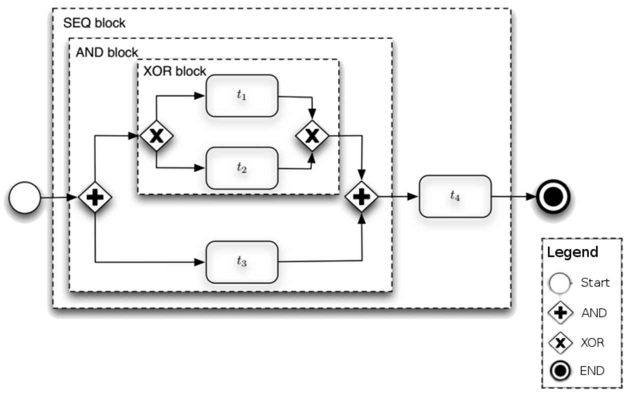

Fig. 1. A Structured Business Process

SBPs consist of *tasks* representing actions performed towards the completion of the process and of *control nodes* structuring the process. Those tasks and control nodes are interconnected using transitions so that the following holds:

- The process has a unique start and end, represented by the Start and End control nodes, so that no transition is incoming to Start or outgoing from End.
- Each task has exactly one incoming and one outgoing transition.
- XOR and AND are control nodes used to delimit blocks representing the nesting of processes. The difference between XOR and AND is purely semantical. While AND means that all sub-processes (outgoing transitions) need to be processed, XOR specifies that exactly one of them has to be processed.

– XOR and AND control nodes have one incoming and more than one outgoing transition if they are used to open a new block (in which case they are called XOR split and AND split), or more than one incoming and one outgoing transition if they are used to close a new block (in which case they are called XOR join and AND join)
– A Block opened by an AND split or XOR split needs to be closed by an AND join or XOR join, respectively.
– The process is acyclic (all tasks are traversed at most once)

An example business process representing a model expressed in this language is represented in fig. 1 using traditional notation from the business process community.

This choice of case study is based on the fact that:

– The SBP's specification has been formalized in [17], thus providing a precise description of the syntax and semantics of this language.
– It has sufficient complexity to illustrate the usefulness of our tool.
– It is practically relevant since many existing business processes are expressible in this form [17].

This case study has been implemented using Lightning in the context of a master thesis. The concrete verification examples presented in this paper have actually been encountered during that work.

3 Lightning

The Lightning tool is a language workbench based on Alloy. It is distributed as an Eclipse plugin [1]. It provides support to formally express all the components of a language (Abstract Syntax, Concrete Syntax, and Semantics), and allows to verify these using Alloy's SAT based model finding mechanism. Amongst the notable features of Lightning are :

– A complete Alloy editor (with outline, error markers and syntax highlighting)
– Ecore support
– An editor allowing to modify generated instances.

The signature trait of Lightning, however, is to allow incremental language development (depicted in fig. 2) by coupling the instance generation of Alloy with the domain specific visualization and model execution induced by the concrete syntax and semantics definition, respectively . This approach facilitates the identification of design errors [7].

In the following sections, we will delve into the details of the design process shown in fig. 2 and describe associated verification tasks.

[1] Freely available at : http://lightning.gforge.uni.lu

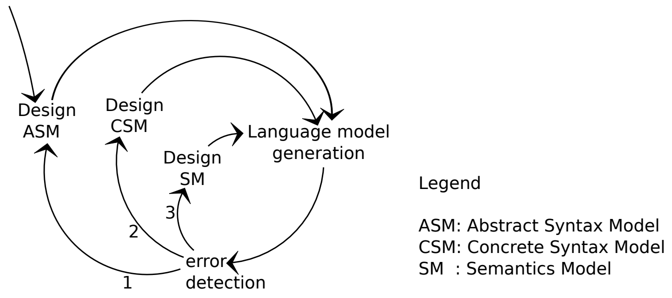
Design ASM
Design CSM
Design SM
Language model generation
error detection
1
2
3
Legend
ASM: Abstract Syntax Model
CSM: Concrete Syntax Model
SM : Semantics Model

Fig. 2. Spiral diagram depicting how languages are incrementally designed in Lightning

4 Abstract Syntax Design

In Lightning the abstract syntax of a language consists of an Alloy model defining the set of valid language models. We can view this model as the *metamodel* of the language. In our SBP case study, the abstract syntax model (ASM) defines concepts of the language (Tasks, Flows, Control nodes, ...), relations between those concepts (e.g., Flows have Nodes as source and target), and well-formedness rules expressed as constraints (e.g., to ensure acyclicity of the process). The following is an excerpt of the abstract syntax model:

```
abstract sig Node{}
abstract sig Control extends Node{}
one sig Start extends Node{}{this not in Flow.target}
one sig End extends Node{}{this not in Flow.source}
sig Task extends Node{}
sig AND_JOIN,AND_SPLIT,XOR_JOIN,XOR_SPLIT extends Control{}
sig Flow{
  source: Node,
  target: Node
}
fact acyclic{
  all n: Node | n not in n.^((~target).source)
}
```

Listing 1.1. Abstract Syntax Model excerpt

We can use Alloy's instance generation mechanism to verify the abstract syntax. This scenario corresponds to the cycle labelled **1** in fig. 2. Figure 3 depicts one of the language models thus obtained from our SBP specification. Although it is still possible to interpret this model correctly, it is a bit tedious since it is not presented in the traditional way but reflects the structure of the abstract syntax. The more complex a language model is (in terms of number of elements and links present), the harder it becomes to comprehend it. This is why it is advised to start defining the concrete syntax of a language (transit to cycle 2 in fig. 2) once its models become hard to check through their default visualization.

In the next section we define how domain specific visualizations are specified in Lightning.

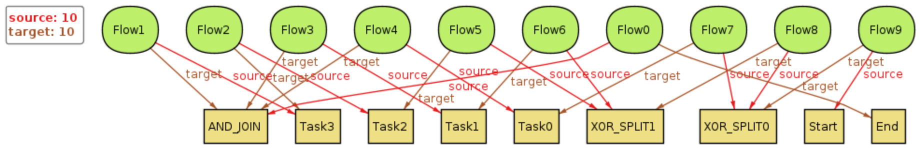

Fig. 3. Raw visualization of a language model (using Alloy's Magic Layout)

5 Concrete Syntax Design

The Concrete Syntax of a language consists of an Alloy model defining a transformation from the previously defined Abstract Syntax Model (ASM) to a predefined Visual Language Model (VLM). This definition follows the approach that Kleppe describes in [13]. This VLM, named LightningVLM and also expressed in Alloy, consists of:

– A set of visual elements that can be linked and composed
– Layout and color declarations that can be used as properties of visual elements
– Well-formedness rules that enforce that any instance can be correctly rendered once interpreted by the tool (by preventing the presence of cyclic compositions, for example)

The transformation model enforces that all of its instances contain a given ASM instance and its corresponding VLM instance via the use of mapping rules and integration predicates; these predicates specify the values of fields of atoms in the VLM instance. The VLM instance can then be interpreted by Lightning in order to be rendered graphically. This process is the essence of the concrete syntax support the tool provides and is depicted in fig. 4. Note that in the current version of Lightning the concrete syntax is used only for visualisation and cannot be directly edited.

In order to be processed in a reasonable time, the Alloy model defining this ASM to VLM transformation can be written following a sub-syntax of Alloy, such that interpretation can be used rather than SAT-solving. This approach called functional module is introduced in [8].

Below we provide a selection of the mapping rules and their integration predicates (prefixed with the prop_ keyword) defined in order to provide a concrete syntax to our SBP language.

```
/* each task is represented by a rectangle, and each node has its
     corresponding label */
one sig Transformation{
     mapTask: Task one -> one RECTANGLE,
     mapNodeText: Node one -> one TEXT
}
/* a task is represented by a rectangle with a white
background that contains the corresponding text */
pred prop_mapTask(n: Task, r:RECTANGLE) {
     r.layout = VERTICAL_LAYOUT
     r.color = WHITE
```

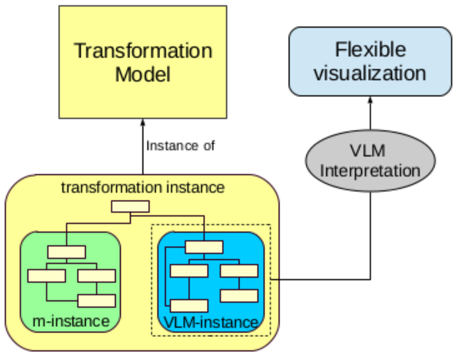

Fig. 4. Visualization process using a transformation from ASM to VLM

```
    r.composedOf[0] = Bridge.mapNodeText[n]
}
/* the text is black, not styled, and is labeled
after the label of the node it represents */
pred prop_mapNodeText(n:Node, t:TEXT) {
    t.color = BLACK
    t.isItalic = False
    t.isBold = False
    t.textLabel[0] = n
}
```

Once the concrete syntax is defined in this way, it becomes easier to detect errors in an instance model. Figure 5 depicts the language model previously shown in fig. 3, visualized this time using its concrete syntax definition.

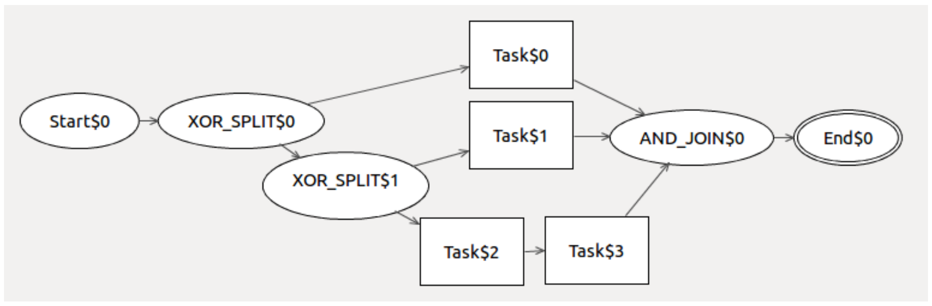

Fig. 5. Visualization of the instance depicted in fig. 3 using its concrete syntax definition

Only one glance at fig. 5 suffices to notice that our SBP language is underspecified. Indeed, in this language model, two XOR splits are converging into a single join. Moreover this join, which is an AND join, doesn't have the same nature than the converging splits. In order to fix this design error, we need to associate splits and joins together. We do this via the definition of control boxes:

```
sig ControlBox {
    split: Control,
    join: Control
}{ (
    // SPLIT AND JOIN HAVE SAME NATURE
    split in AND_SPLIT and join in AND_JOIN) or
   (split in XOR_SPLIT and join in XOR_JOIN)
    // PAIRING EACH SPLIT WITH A GIVEN JOIN
    all s: (succ[split]) | s in (preds[join])
    all j: (pre[join]) | j in (succs[split])
}
```

Adding the concept of a control box to the abstract syntax and repeating the instance generation shows us that the error has been well identified and fixed. The error processing we just discussed illustrates a transit to the cycle 1 of fig. 2., i.e., to the case where an error found in the visualisation reveals an error in the underlying abstract syntax. Of course the transformation model describing the visualisation may be faulty itself. In this case the error in the visual representation may point to an error in the concrete syntax model. This situation corresponds to a transit to the cycle 2 of fig. 2, leading to redesigning the Concrete Syntax model. Checking if the error seen in the concrete syntax visualization is also present in the concrete-syntax-less visualization (described in the previous section) allows to decide whether or not the error has been introduced by the concrete syntax definition.

6 Semantics Definition

Lightning currently offers the possibility to define the operational semantics of languages.

The semantics definition in Lightning consists of:

- a Semantics Model (SM) in which the concepts of *state* and *trace* are defined. A step predicate is specified that expresses the condition that one state follows another state in the trace.
- a Semantics visualization transformation model, reusing most of the rules present in the ASM to VLM transformation but adding rules to represent the properties of the semantics state.

For our case study each state consists of a set of nodes that are currently active in the execution of the business process. The corresponding field of the Alloy signature is called *currentNodes*. That is, for a given state s, the expression *s.currentNodes* denotes the set of active nodes in state s. The visualisation represents the currently active nodes by highlighting them in the business process model.

To verify the correctness of the operational semantics, one can visualize its possible executions. To illustrate this verification, let us consider the following predicate as a first attempt to define the semantics of XORs:

```
pred XORNodes(current:Node,s2:State) {
    current in XOR_SPLIT  and one node: current.(~source).target | node
        in s2.currentNodes
}
```

This predicate ensures that given a current node that is a XOR_SPLIT, the set of current nodes belonging to the next semantics state contains exactly one of the nodes directly following the XOR_SPLIT (mutual exclusion). Figure 6 gives an example of an erroneous execution.

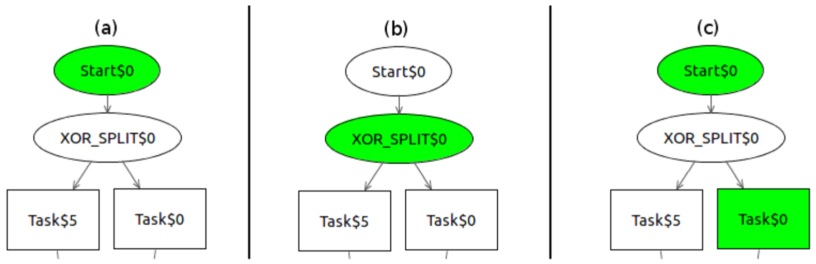

Fig. 6. Erroneous execution of a business process model

Although the transition from (a) to (b) is performed as expected, the transition from (b) to (c) shows us that our XOR semantics is underspecified. Indeed, the predicate previously shown enforces that only one of the nodes directly following an active XOR_SPLIT should be part of the current nodes. This predicate thus does not specify the state of the other nodes, thus allowing extraneous nodes to appear in the set of current nodes for a given state. To fix this, one simply needs to enforce that the set of current nodes of a given state is contained in the set of successors of all the current nodes present in the previous semantics state (code omitted for lack of space).

The example above illustrates the case where an error in the concrete syntax representation of the semantic state points to an error in the underlying semantic model. This case corresponds to the cycle 3 of fig. 2.

7 Discussion and Related Work

The term "language workbench" was made popular by Martin Fowler [6]; it denotes a tool that supports the efficient definition, reuse and composition of languages and their IDEs [5]. The Lightning tool may be viewed as a language workbench that is based on the formal language Alloy (although not a full-fledged one as mentioned in the introduction). Because of its formal basis it differs from existing language workbenches such as MetaEdit+[11], MPS[18], and Spoofax[10]. Few workbenches currently support formal semantic analysis; notable exceptions are Kermeta [2] and Atom 3 [4] for which some formal analysis is available via a translational semantics (to Maude for Kermeta [3] and to Alloy for AToM3 [2], [19]).

Our work is based on the premise that developing modelling languages benefits from the lightweight formal modelling approach offered by Alloy because it gives language developers immediate feedback on design decisions using automatic formal analysis and thus allows to detect design errors early. We can thus view Lightning as an attempt to provide agile modelling of software languages in a way similar to the initial intent of Alloy, namely providing agile modelling of software designs.

Because our tool is based on Alloy it also inherits the inherent limitations of Alloy. Indeed verification is based on instance finding via SAT solving. The effectiveness of this approach intimately depends on the small scope hypothesis, stating that most of the design errors can be found in small models. Assuming the small scope hypothesis holds, the approach will allow to reduce the scopes of signatures in Alloy so that a correct answer can be found in reasonable time. Of course a negative answer in the search of a counterexample does not exclude the possibility that there may be one but may point instead to the need for trying out larger scopes, resulting of course in longer running times.

In the context of language design, though every aspect of a language is written in Alloy, the performance limitations of Alloy we just mentioned only apply to the generation of language models (ASM instances). The visualisation and semantics, benefiting from functional modules, can be processed efficiently [8].

8 Conclusion

We have presented in this paper how Lightning allows the application of a lightweight verification technique based on Alloy from the earliest stages of a domain specific language design process to its completion. In particular we have given concrete examples of verification tasks that were carried out during the design of a language for structured business processes.

Regarding future work much remains to be done. One obvious hindrance to the use of our tool is the fact that it requires prior knowledge of Alloy. We are currently trying to see to what extent we can provide graphical interfaces to most of the modelling tasks in the tool. In particular we have already partially implemented such an interface for defining transformations.

Another fundamental question that needs to be investigated concerns performance. Indeed, once the metamodel becomes a bit larger (with, say, tens of signatures) Alloy's instance generation tends to slow down appreciably. Recent work on model slicing (such as [14, 15]) in the context of UML/OCL models) suggests that in many cases instance generation can be made more efficient by generating instances for subparts of the metamodel and then combining these partial instances into an instance of the whole metamodel. We plan to investigate this type of approach in the context of our work.

[2] A newer version of the tool named AToMPM [16] is available

References

1. Lightning tool web site, http://lightning.gforge.uni.lu.
2. Kermeta tool web site, http://www.kermeta.org.
3. Moussa Amrani. A formal semantics of kermeta. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, 2012.
4. Juan De Lara and Hans Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In *Fundamental approaches to software engineering*, pages 174–188. Springer, 2002.
5. Sebastian et al. Erdweg. The state of the art in language workbenches. In Martin Erwig, Richard F. Paige, and Eric Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer International Publishing, 2013.
6. Martin Fowler. Language workbenches: The killer-app for domain specific languages. http://martinfowler.com/articles/languageWorkbench.html.
7. Loïc Gammaitoni and Pierre Kelsen. Domain-specific visualization of alloy instances. In *ABZ*, pages 324–327, 2014.
8. Loïc Gammaitoni and Pierre Kelsen. Functional Alloy Modules. Technical Report TR-LASSY-14-02, University of Luxembourg; http://hdl.handle.net/10993/16386.
9. Daniel Jackson. *Software abstractions.* MIT Press Cambridge, 2012.
10. Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010.
11. Steven Kelly, Kalle Lyytinen, and Matti Rossi. Metaedit+ a fully configurable multi-user and multi-tool CASE and CAME environment. In *Advanced Information Systems Engineering*, pages 1–21. Springer, 1996.
12. Pierre Kelsen and Qin Ma. A lightweight approach for defining the formal semantics of a modeling language. In *Model Driven Engineering Languages and Systems*, pages 690–704. Springer, 2008.
13. Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels.* Addison-Wesley Professional, 2008.
14. Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. Verification-driven slicing of uml/ocl models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 185–194. ACM, 2010.
15. Asadullah Shaikh, Uffe Kock Wiil, and Nasrullah Memon. Uost: Uml/ocl aggressive slicing technique for efficient verification of models. In *System Analysis and Modeling: About Models*, pages 173–192. Springer, 2011.
16. Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In *Demos/Posters/StudentResearch@ MoDELS*, pages 21–25, 2013.
17. Silvano Colombo Tosatto, Guido Governatori, and Pierre Kelsen. Towards an abstract framework for compliance. *Proceedings of the 17th IEEE International EDOC 2013 Conference Workshops*, pages 79–88, 2013.
18. Markus Voelter and Vaclav Pech. Language modularity with the mps language workbench. In *34th International Conference on Software Engineering (ICSE)*, pages 1449–1450. IEEE, 2012.
19. Thomas De Vylder. Feature modelling: A survey, a formalism and a transformation for analysis. University of Antwerp.