SLE 2015: Tool Paper submission

Designing Languages using Lightning

Loïc Gammaitoni
University of Luxembourg
loic.gammaitoni@uni.lu

Pierre Kelsen
University of Luxembourg
pierre.kelsen@uni.lu

Christian Glodt
University of Luxembourg
christian.glodt@uni.lu

Abstract

Modelling languages are defined by specifying their abstract syntax, concrete syntax and semantics. In the Lightning tool the definition of all these language components is based on the lightweight formal language Alloy. Lightning makes use of the powerful automatic analysis features of Alloy to allow language designers to develop and validate the definition of a modelling language in an incremental fashion. By providing immediate visual feedback, it allows errors in the language definition to be quickly identified and corrected. Furthermore Lightning introduces a novel interpretation mechanism that allows efficient execution of transformations used in the language definition. We illustrate the use of the tool on the language of structured business processes.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Computer-aided software engineering

General Terms Design, Verification

Keywords Language Design, Alloy , Language Workbench, Lightweight Formal Method, Agile Design, Lightning

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]. It is termed a lightweight formal language because of its fully automated analysis features. These features allow software modellers to develop software designs in an incremental fashion with immediate visual feedback.

The Lightning tool [1] aims at carrying over the automated analysis features of Alloy to the domain of software language design. The suitability of Alloy for specifying the different syntactic and semantic aspects of a language has already been studied earlier in [12]. The present paper may be viewed as a concrete validation of these ideas.

The Lightning tool, in its current form, may be considered as a prototypical language workbench. Compared to existing language workbenches it lacks sophisticated editor support and code generation facilities. On the other hand it innovates by providing advanced verification capabilities for all main components of a language definition – abstract syntax, concrete syntax and semantics.

Two major challenges have to be dealt with when creating an environment based on a formal language: first, the difficulty of writing formal specifications has to be taken into account; secondly, performance issues (that may hinder a practical adaptation) have to be considered. Regarding the first challenge we observe that the automatic and continuous validation of language models [1] via Alloy's analysis as well as the object-oriented nature of the language and its tiny core based on the notion of a mathematical relation facilitate the writing of language definitions. As for the second challenge the performance issue was particularly problematic when writing transformations (the combinatorial size of transformation models increases drastically the analysis time complexity). Lightning makes use of a sublanguage of Alloy, named F-Alloy[7], that allows transformations to be written using syntax compatible with Alloy, but also to be efficiently interpreted (rather than analysed).

The remainder of this paper is structured as follows: in the next section we give general information on the tool architecture. In section 3 we introduce the structured business process language (SBP) as a case study to illustrate our approach. In section 4 we describe which models compose a language definition in Lightning, and motivate those design choices. In section 5 we clarify where analysis and interpretation are used and how the tool benefits from the union of those two technologies. In section 6, we give an example of the incremental design cycle by showing how the SBP language introduced in section 3 can be specified. We then compare Lightning features to those of some fully-fledged language workbenches in section 7. In the final section we present concluding remarks and future work.

[1] We use the term language model throughout the paper to mean a model conforming to a language.

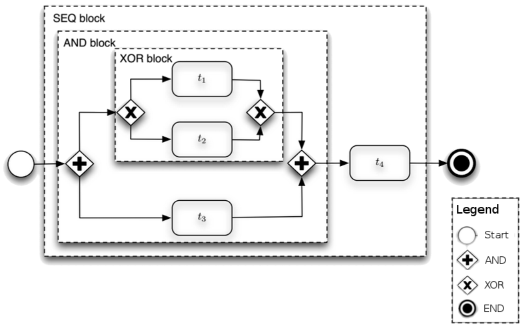


Figure 1: A Structured Business Process

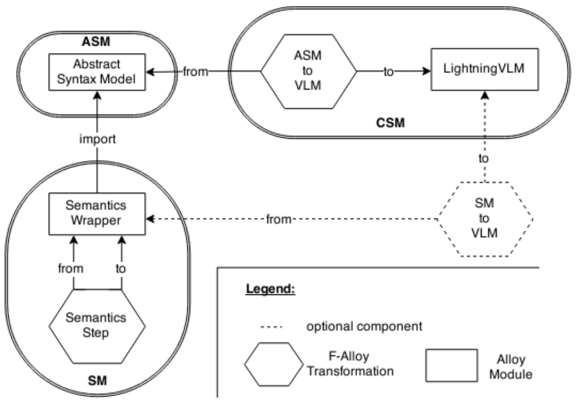


Figure 2: Representation of languages in Lightning

2. Tool Architecture

The Lightning tool is a java 1.6-compliant environment distributed as an Eclipse plugin [2]. It relies on

- Alloy 4.2 for the analysis and syntactic validation of Alloy Models.
- UML2Alloy, UML2, OCL4KMF, oclxmi2Ecore and dresden OCL lib for Ecore 2 Alloy and language model to XMI conversions.
- Draw2D for all the visualizations rendering.

3. Case Study

In this paper, we illustrate language design using Lightning with the help of a concrete language, the Structured Business Process (SBP) language[3]. This case study was first presented in [8].

Structured business processes 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 [14], 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 [14].

4. Language Definition

The Lightning tool allows the definition of languages following the approach proposed by Kleppe[13]. Hence, in Lightning, a language definition consists of an Abstract Syntax Model (ASM) specifying the set of valid language models, a Concrete Syntax Model (CSM) defining the relations between language models and their graphical representations, and a Semantics Model (SM) providing a meaning to those models. In this section, we detail how those aspects are represented in the Lightning tool.

4.1 Abstract Syntax

An ASM consists, in Lightning, of an Alloy model that formally defines the concepts, inter-concept relations, and constraints of a language. It determines the set of valid language models and plays a central role in the language definition as shown in fig.2. In the remainder of this section

[2] Official website: http://lightning.gforge.uni.lu

[3] The full implementation in Lightning of this language can be found at http://lightning.gforge.uni.lu/examples/SBP.zip

we explain how the concrete syntax and semantics both depend on the abstract syntax.

Two features related to abstract syntax are accessible to Alloy neophytes: the first feature allows import of abstract syntax specifications expressed as Ecore models (which may contain OCL constraints), which are then translated into corresponding Alloy models. These models can then be analysed and the obtained instances can be exported (using the second feature) as Ecore instances.

4.2 Concrete Syntax

The Concrete Syntax support provided by the Lightning tool is limited to the visualization of language models. Thus, the tool does not allow editing language models by directly manipulating their concrete representation (this is further discussed in sec.7).

The visualization of language models is formally defined as a model transformation, expressed in F-Alloy[7], from the ASM to the Lightning Visual Language Model (LightningVLM).

F-Alloy is a sublanguage of Alloy in the sense that every module expressed in F-Alloy is also a valid Alloy module. F-Alloy has a translational semantics, allowing transformations to be expressed more succinctly than if we were using plain Alloy. The main reason for defining a sublanguage for Alloy is the possibility of efficiently interpreting (rather than analysing) modules expressed in F-Alloy. Interpretation will be discussed in more depth in the next section.

This visualisation transformation is represented by a hexagonal shape in the upper right corner of figure 2. LightningVLM consists of :

- a set of visual elements that can be connected to each other and composed
- layout and color declarations that can be used as properties of visual elements
- well-formedness rules enforcing that any instance can be correctly rendered once interpreted by the tool (e.g., by preventing the presence of cyclic compositions)

The VLM instance, resulting from this transformation, can then be interpreted by Lightning in order to be rendered graphically. Further information about this feature can be found in [6].

4.3 Semantics

The Lightning tool currently supports the definition of operational semantics of a language via two artefacts :

- A semantic domain model, taking the form of an Alloy module importing the ASM, that aims at adding the concept of state to the abstract syntax. It can also contain functions – i.e. set-valued parametrized expressions – defining a succession relation between states as well as a predicate – i.e. boolean-valued parametrized expression – defining the conditions under which an execution state can be initial.
- A semantic step transformation, taking the form of an F-Alloy endogenous model transformation – from semantic domain model to semantic domain model – that defines how to obtain the next state in the execution of a language model.

Additionally, as displayed in dashed lines in fig.2, the semantics model can be accompanied by an extra F-Alloy transformation from the semantic domain model to LightningVLM, generally extending the ASM to VLM transformation , so that the execution of a language model according to its operational semantics can be visualised.

5. Analysis versus Interpretation

In this section, we give more details on the key mechanisms that are used by the Lightning tool, namely the analysis of Alloy models and the interpretation of F-Alloy transformations.

5.1 Analysis

The analysis carried out by the Alloy Analyzer is integrated in the Lightning tool and allows, given a command defining a scope [4], to generate a finite set of instances from Alloy models. It is used to generate language models from ASMs. The time complexity of such analysis depends on the scope given in the command. According to the small scope hypothesis [9] small scopes will generally suffice to detect most design errors.

Analysis is also used for calculating the initial state of a model when executing it, i.e., when applying its operational semantics. Calculation of the initial state is a two step process: first a given language model is converted into an Alloy model which is constrained so as to admit only this particular language model as unique instance. Next, the semantic domain model is analysed, taking into account the previously constrained model, to determine the initial state for the given language model. This analysis is usually efficient since the language model itself is fixed and only the state portion (specified in the semantic domain model) needs to be analysed.

5.2 Interpretation

Transformations expressed in F-Alloy can be computed efficiently in a backtrack-free manner, as described in [7]. This is essential to the operation of the Lightning tool. Indeed, executing transformations via analysis has proven to be very time consuming thus making it impractical for visualisation for instance. The extra time complexity for analysing transformations is not surprising since transformations involve

[4] an upperbound on the number of atoms composing instances to be generated.

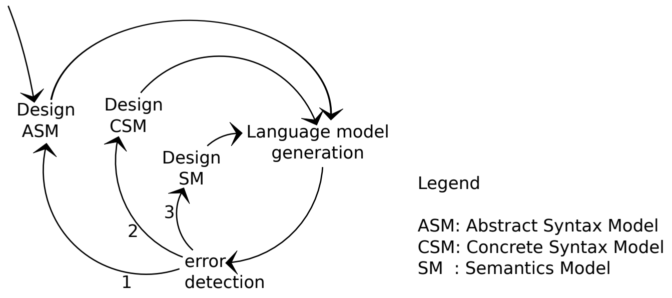

Design ASM
Design CSM
Design SM
Language model generation
error detection
1
2
3
Legend
ASM: Abstract Syntax Model
CSM: Concrete Syntax Model
SM : Semantics Model

Figure 3: Spiral diagram depicting how languages are incrementally designed in Lightning

two metamodels, the input and output metamodels, both represented by Alloy models.

Interpretation of transformations expressed in F-Alloy is done in several places:

- to visualise language models using their concrete syntax.
- to execute a language model by repeating the semantic step transformation
- to visualise the execution of language models
- to compute simple language to language (ASM to ASM) transformations (see [7])

6. Language Design Cycle

In this section, we describe how Lightning can be used to design languages in an incremental (see fig. 3) manner. We illustrate this approach using the case study introduced in sec. 3. The main focus of this section will be the design of the language semantics. Indeed the process of defining the abstract and concrete syntax has already been described in [8] [5] ; it will only be sketched in the next subsection.

6.1 Abstract and Concrete Syntax Design

As mentioned earlier the abstract syntax model (ASM) is represented by an Alloy model defining the concepts and constraints of the language. Here is an excerpt of the SBP abstract syntax:

```
module SBP/AbstractSyntax/ASM
abstract sig Node{}{
  this not in successors[this] // no cycles
  this in successors[Start] + Start //reachable
}
sig Task,End,Start extends Node {}
abstract sig Control extends Node {}
sig AND_JOIN,XOR_JOIN,AND_SPLIT,XOR_SPLIT
        extends Control {}
sig Flow{
  source: Node,
  target: Node
}
fun nextNodes(n: set Node) : set Node {
        n.(~source).target
}
fun successors(n: set Node) : set Node {
        n.^((~source).target)
}
```

[5] [8] also presents an analysis based approach to semantics, differing from the transformation based approach described here

At any stage of defining the abstract syntax we can debug the current description by generating instances for the Alloy model. Some errors can be spotted and corrected in this way (cycle 1 in fig. 3). In general however it will be easier to debug the abstract syntax if we can view the instances Lightning produces using the concrete syntax. If we detect an error with the help of the concrete syntax representation, the error may be of two kinds: either an error in the abstract syntax or an error in the concrete syntax itself. Fixing an error of the first kind corresponds to passing into cycle 1; errors of the second type correspond to a passage into cycle 2. For more details the reader is referred to [8].

6.2 Semantics Design

As mentioned earlier, Lightning allows the specification of operational semantics. The first step in semantics design is thus to identify what notions are missing in the ASM to express a step of language execution. Our SBP language as defined during the abstract syntax design phase induces a node structure with a start node, an end node, and precedence relations in the form of flows. But to model an execution step, the notion of active node is missing. Indeed, nothing in the ASM allows us to define which node is active in a given state of execution. Once the missing concept is identified, it is added to the ASM in what we call a semantic domain model. Here is an excerpt of the semantic domain model associated to the SBP language. Note the presence of the `init` predicate that ensures that in the initial state only the Start node is active.

```
module SBP/Semantics/Semantics
open SBP/AbstractSyntax/ASM

one sig State{
  activeNodes: some Node
}
pred init [s:State] {
  s.activeNodes = Start
}

run init
```

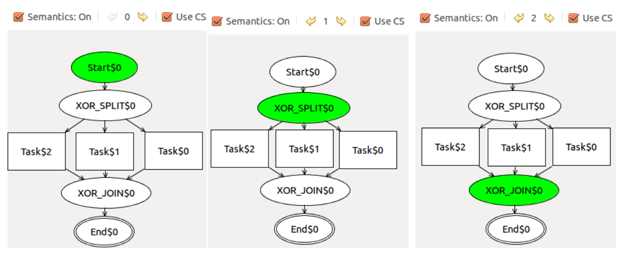
Given this semantic domain model, it is possible to model a step of language execution as an endogenous transformation from the semantic domain model to itself.

In our SBP example, the transformation is described as follows :

```
module SBP/Semantics/Semantics
open SBP/Semantics/Semantics

one sig Bridge{
  map: State -> State
}

pred guard_map(s: State) {
  not s = End.(~activeNodes)
}
pred value_map(s1: State, s2: State) {
  s2.activeNodes = nextActiveNodes[s1.activeNodes]
}
```



(a) starting state (b) state 1 (c) state 2

Figure 4: Faulty semantics execution of an SBP language model containing XORs

Note that endogenous transformations in F-Alloy conserve elements that are not part of a mapping. Hence, this transformation only enforces that the state of execution of a given semantic domain model instance should be replaced (if the state's active node is not an End) by a new execution state where the active nodes are given by the function `nextActiveNodes[n:Node]`. This helper function is defined in the semantic domain model as it is meant to be evaluated in a semantic domain model instance and has been defined as follows :

```
fun nextActiveNodes(n: set Node): set Node {
  //return nextNodes in general except :
  nextNodes[n - AND_JOIN - XOR_SPLIT] +
  //if and_join: wait no nodes in n are preceding it
  nextNodes[{x: n & AND_JOIN | n & predecessors[x]=none}]+
  //if xor_split: return solely one successor .
  {x: Node | some y: n & XOR_SPLIT |
        x= order/max[successors[y]]}
}
```

An execution of this semantics definition leads to the sequential visualisation depicted in figure 4. This execution reveals an error since one of the task nodes needs to become active between the activation of `XOR_SPLIT` and `XOR_JOIN`.

This visualisation thus points to an error in the `nextActiveNodes` function for XOR control nodes. A closer look at the XOR related specifications allows us to realize that the error is due to an ambiguous naming of functions. Indeed, the `successors` function defined in the ASM returns the set of all nodes succeeding the nodes given in parameter. Replacing the call to the function `successors` by a call to the `nextNodes` function, which only returns direct successors of the nodes given in parameter, will fix the error.

7. Discussion

The term "language workbench" was made popular by Martin Fowler [5]; it denotes a tool that supports the efficient definition, reuse and composition of languages and their IDEs[3]. While Lightning is not yet a full-fledged language workbench (as we will see below) it clearly aims at facilitating the development of new (domain-specific) languages. To understand what Lightning brings to the software language engineering research, it is therefore instructive to analyse its features in the context of existing language workbenches. We base our comparison on the domain analysis performed by [3] which resulted in table 1. In this table we have also indicated the features of the Lightning tool.

Features	Language Workbench		Lightning
	mandatory	optional	
Notation	✓	✗	graphical
Semantics	✓	✗	operational
Editor Support	✓	✗	tree editor
Syntactic services	✗	✓	✗
Semantics services	✗	✓	✗
Validation	✗	✓	Alloy Validation
Formal Verification	✗	✗	Alloy Verification
Testing	✗	✓	exhaustive testing
Composability	✗	✓	only syntactic

Table 1: Comparative table of Language workbench features(as proposed in [3]) and of Lightning features

The notation that Lightning uses for presenting models to the user is graphical: models are either shown in a concrete syntax notation (defined by a transformation of the abstract syntax model to a visual language) that is not editable, or in the form of a tree representation that can be edited.

The only editor support that Lightning currently offers is a tree based editor, supplemented by a graphical view of the model being edited (based on the concrete syntax of the language). Compared to mature language workbenches Lightning is lacking comfortable editor support. This is shown in the lack of syntactic and semantic editor services. In contrast, the language workbenches MetaEdit+[11], MPS[15], Spoofax[10], and Xtext[4] offer the full range of syntactic editor services – highlighting, outline, folding, syntactic completion, diff, and auto formatting. These same language workbenches also offer a range of semantic editor service such as reference resolution, semantic completion, refactoring and error marking.

While many existing workbenches offer advanced editor support, features that support validation and testing are less commonly found. Lightning offers at this point limited validation of models: upon saving a model that does not conform to the language a textual error message is shown.

The strength of Lightning lies in the validation mechanisms at the level of the language definition which is not an expected feature of Language workbenches according to [3]. At the level of the language definition Lightning offers advanced syntactical validation via Alloy. More importantly Alloy's automatic analysis verifies the consistency of the language definition by generating sample instances. If no instances can be found, the abstract syntax definition is inconsistent. A similar validation can be carried out at the level of the semantics definition. For both syntax and semantics the visualisation based on the concrete syntax definition aids in understanding the generated instances and the ensuing problems in the current specification.

Although some workbenches such as MPS offer support for testing at the level of the language definition, the SAT based analysis of Alloy is arguably more complete since it exhaustively tests all instances up to a given size. Assuming that the small scope hypothesis which Alloy is based on - stating that errors usually admit small counter examples - does indeed hold for language definitions, then the Lightning tool will indeed uncover problems in the language specification.

Regarding the last category of features, composability, Lightning at this point offers mostly syntactic composition via module imports in Alloy. Other workbenches such as Metaedit and Spoofax offer a complete set of composability features covering all aspects of a language: syntax, validation, semantics, and editor services.

There are two more aspects that need to be considered and that are not covered by the features in table 1: formality and the underlying paradigm. By formality we mean the existence of a formal/mathematical foundation for the language workbench. The formal nature of Lightning is a distinguishing characteristics. The workbenches reviewed in [3] do not have a direct formal foundation. The ATOM3 tool[2] (mentioned in [3]) has a formal basis in graph transformations.

Another aspect is the basic paradigm underlying the tool. A distinguishing feature of Lightning is its model-driven nature: the abstract syntax, concrete syntax and semantics represent metamodels which the generated instances will conform to. Most of the language workbenches reviewed in [3] use other means of specifying languages (e.g. grammars), a notable exception being Metaedit+ which is also fully model-based.

8. Conclusion and Future Work

The tool presented in this paper provides an Alloy-based approach to software language engineering. Two basic mechanisms are implemented in the tool: on one hand the automated analysis of Alloy models with its immediate visual feedback permits quick detection of design errors; on the other hand, the interpretation of modules written in F-Alloy allows an efficient computation of transformations in the tool, thus making for instance the specification of visualisation transformations in Alloy (via F-Alloy) practical.

Future work will explore two main avenues: on one hand we would like to open the use of the tool to users not familiar with Alloy. This is already possible to some extent by importing an Ecore specification of an abstract syntax and exporting associated instances. We will explore other ways to reduce the onus of writing specifications. One could envisage for instance that transformations could be designed largely graphically, with only small text snippets needing to be written at some points.

Another avenue of research concerns the use of Lighning as a language workbench. Further editor support as well as code generation capabilities have to be provided to broaden the applicability of the tool. This will also be the occasion to explore the scalability of the tool on larger case studies.

References

[1] Lightning tool web site, http://lightning.gforge.uni.lu.

[2] De Lara, J., and Vangheluwe, H. Atom3: A tool for multi-formalism and meta-modelling. In *Fundamental approaches to software engineering*. Springer, 2002, pp. 174–188.

[3] Erdweg, S. e. a. The state of the art in language workbenches. In *Software Language Engineering*, M. Erwig, R. F. Paige, and E. Wyk, Eds., vol. 8225 of *Lecture Notes in Computer Science*. Springer International Publishing, 2013, pp. 197–217.

[4] Eysholdt, M., and Behrens, H. Xtext: implement your language faster than the quick and dirty way. In *Proceedings*. 2010, pp. 307–309.

[5] Fowler, M. Language workbenches: The killer-app for domain specific languages.

[6] Gammaitoni, L., and Kelsen, P. Domain-specific visualization of alloy instances. In *ABZ*. 2014, pp. 324–327.

[7] Gammaitoni, L., and Kelsen, P. F-Alloy: an Alloy Based Model Transformation Language. In *International Conference on Model Transformations*. 2015, to appear.

[8] Gammaitoni, L., Kelsen, P., and Mathey, F. Verifying modelling languages using lightning: a case study. In *11th Workshop on Model Design, Verification and Validation Integrating Verification and Validation in MDE (MoDeVVa 2014)*. 2014, pp. 19–28.

[9] Jackson, D. *Software abstractions*. MIT Press Cambridge, 2012.

[10] Kats, L. C., and Visser, E. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *ACM Sigplan Notices*, vol. 45. 2010, pp. 444–463.

[11] Kelly, S., Lyytinen, K., and Rossi, M. Metaedit+ a fully configurable multi-user and multi-tool CASE and CAME environment. In *Advanced Information Systems Engineering*. 1996, pp. 1–21.

[12] Kelsen, P., and Ma, Q. A lightweight approach for defining the formal semantics of a modeling language. In *Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 690–704.

[13] Kleppe, A. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

[14] Tosatto, S. C., Governatori, G., and Kelsen, P. Towards an abstract framework for compliance. IEEE Computer Society, Los Alamitos, CA, USA, 2013, pp. 79–88.

[15] Voelter, M., and Pech, V. Language modularity with the mps language workbench. In *34th International Conference on Software Engineering (ICSE)*. 2012, pp. 1449–1450.

APPENDIX: Lightning Tool Demo : How to Define a Structured Business Process Language with Lightning

Loïc Gammaitoni, Pierre Kelsen, and Christian Glodt

University of Luxembourg

Introduction

In this appendix, we present the content and structure of the demonstration we propose to give in SLE 2015. Our demonstration will follow a step by step approach with a Structured Business Process language as supporting case study. We will provide an overview of the main features offered by Lightning, including:

– Language definition
– Language verification
– Language model edition
– semantics execution

Step 1: Language Definition

The presentation will start with a brief introduction of Lightning and of the chosen case study: the design of a Structured Business Process language. Then, we will show how languages are structured in Lightning, in theory (fig.1a) and in the tool (fig. 1b).

We also identify two kinds of modules: ".als"() files represent regular Alloy file, while ".fals"() files represent F-Alloy modules.

Step 2: ASM Design

We propose an SBP abstract syntax design to the audience (fig 2 shows the alloy code and the associated graphical view), and show how to verify it's correctness using the tool by generating several model instances using Alloy analysis.
We will observe that ASM instances can directly be used to debug such specifications. Indeed we notice that some tasks are not connected to any flows (fig.3). After correcting this error we will also note that the unintuitive visualisation of abstract syntax instances may stand in the way of efficient verification.

Step 3 : Use of Concrete Syntax

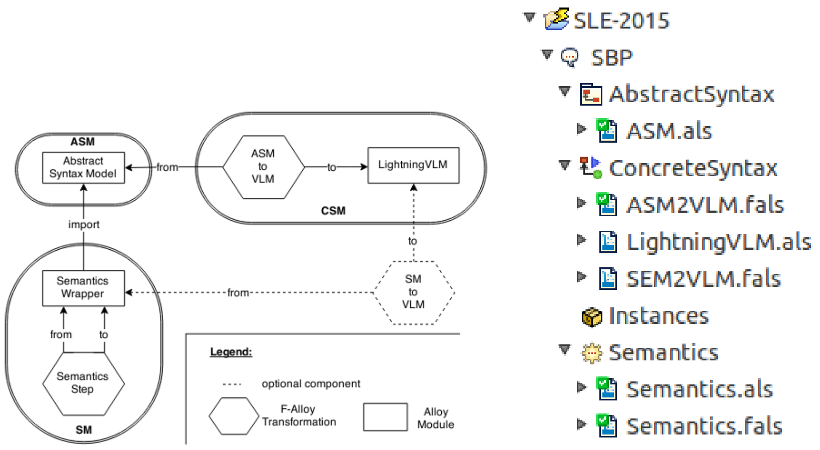
We specify here from scratch an F-Alloy transformation model that specifies the concrete syntax of the language and explain its structure.
This can be performed seamlessly with the predicate auto-generation feature of the Lightning F-Alloy editor.
Once the transformation is specified we generate and visualise several instances of our SBP language to realise that the ASM definition still contains error (fig. 4 suggests indeed that the control nodes are underspecified).
After fixing the error we visualise several instances one last time to increase our confidence in the correctness of the ASM (fig.5).
We then save one of the instance displayed to edit it in the next section.

Step 4 : Edit Instances

We open the previously saved instance using the instance editor.
We notice that an instance viewer is open at the same time than the editor. Each modification brought to the instance is directly reflected to the visualisation and a colourful indicator will let the user know if the instance obtained after modification is conforming to its metamodel (see fig.6 and 7).

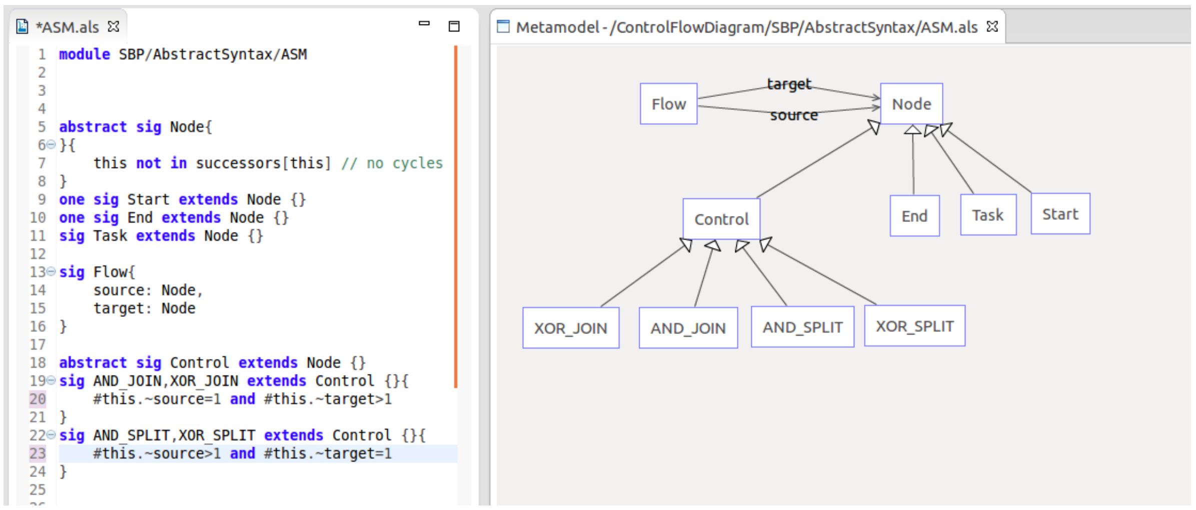
Step 5 : Define, Execute and Verify Semantics

Proceeding to the definition of semantics, we import three models to our language: a semantic domain model, a semantic step transformation and a semantic visualization transformation. After going briefly through them while highlighting their purpose, we execute a language model (obtained by analysis). The execution is depicted in figure 8, and indicates an error in the behaviour of `XOR_SPLIT` control nodes. After explaining the cause of this error, and fixing it, we execute several other language models obtained via analysis to increase our confidence in the correctness of our language definition.



(a) Theoretical Overview (b) in-tool view

Fig. 1: Language Structure in Lightning



```
module SBP/AbstractSyntax/ASM



abstract sig Node{
}{
    this not in successors[this] // no cycles
}
one sig Start extends Node {}
one sig End extends Node {}
sig Task extends Node {}

sig Flow{
    source: Node,
    target: Node
}

abstract sig Control extends Node {}
sig AND_JOIN,XOR_JOIN extends Control {}{
    #this.~source=1 and #this.~target>1
}
sig AND_SPLIT,XOR_SPLIT extends Control {}{
    #this.~source>1 and #this.~target=1
}
```

Fig. 2: Alloy specification of the SBP language's abstract syntax and its associated graphical metamodel view

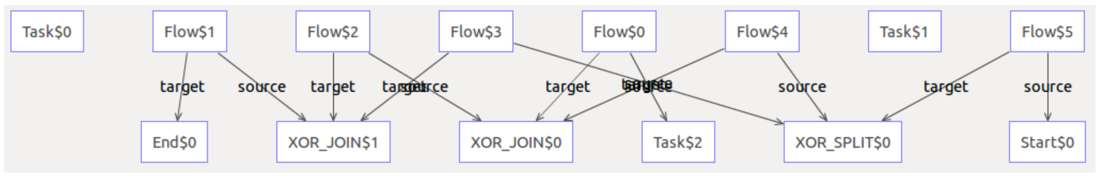


Fig. 3: Instance visualization highlights that tasks may not be connected

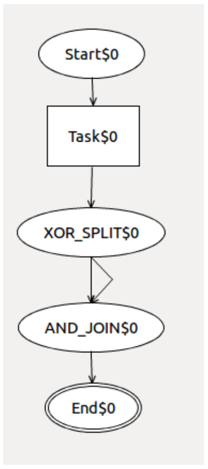


Fig. 4: Instance visualization with concrete syntax applied highlights design error in the control specifications

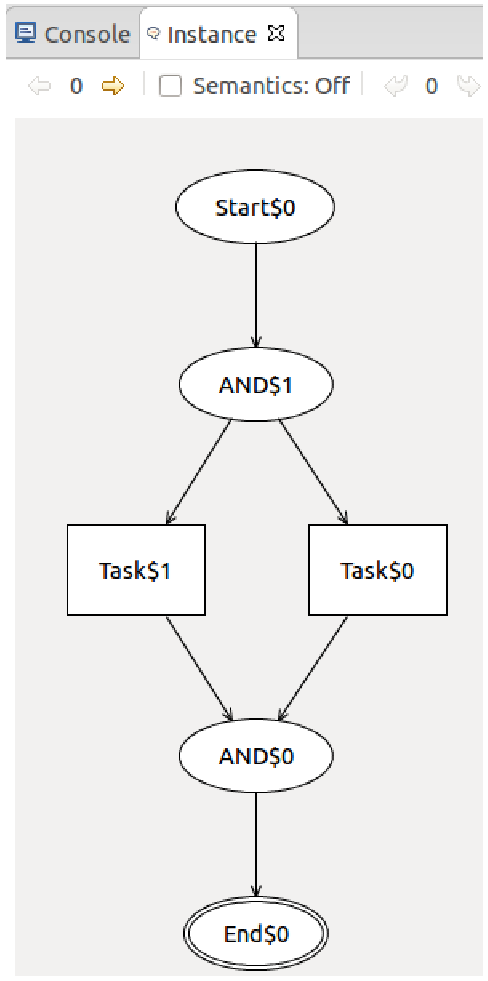
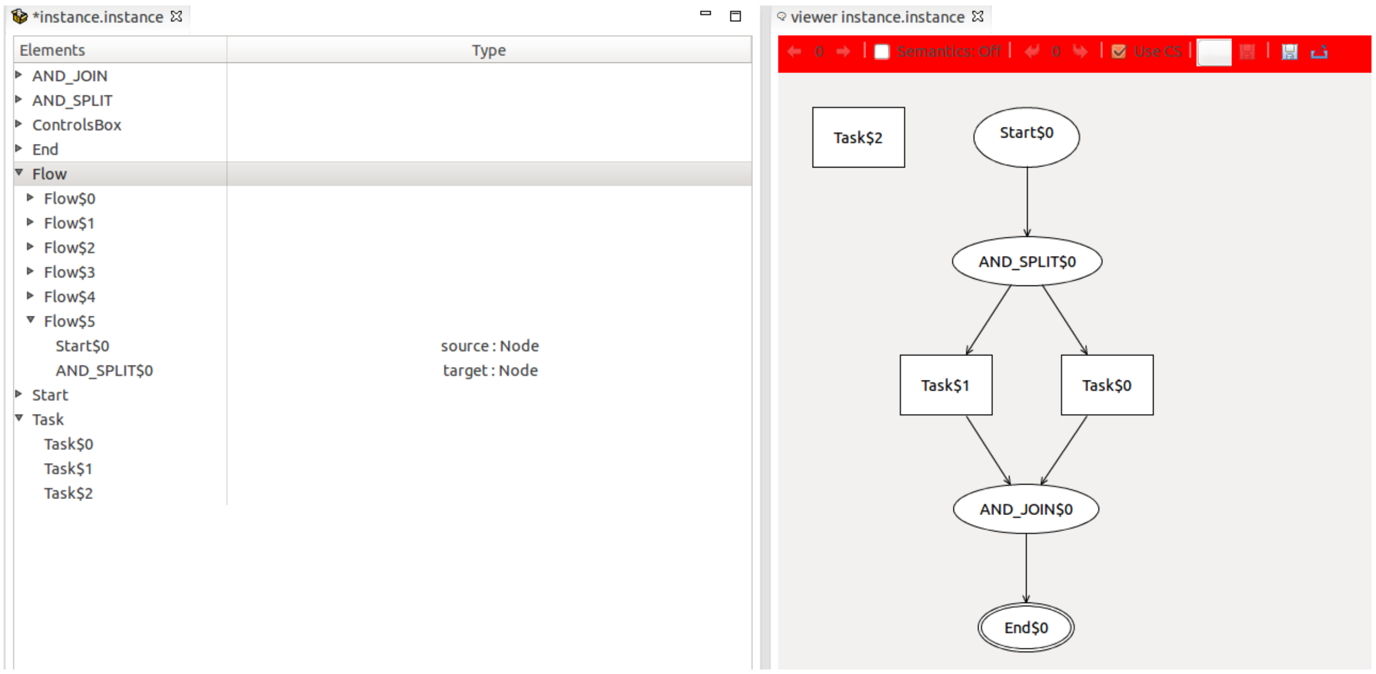
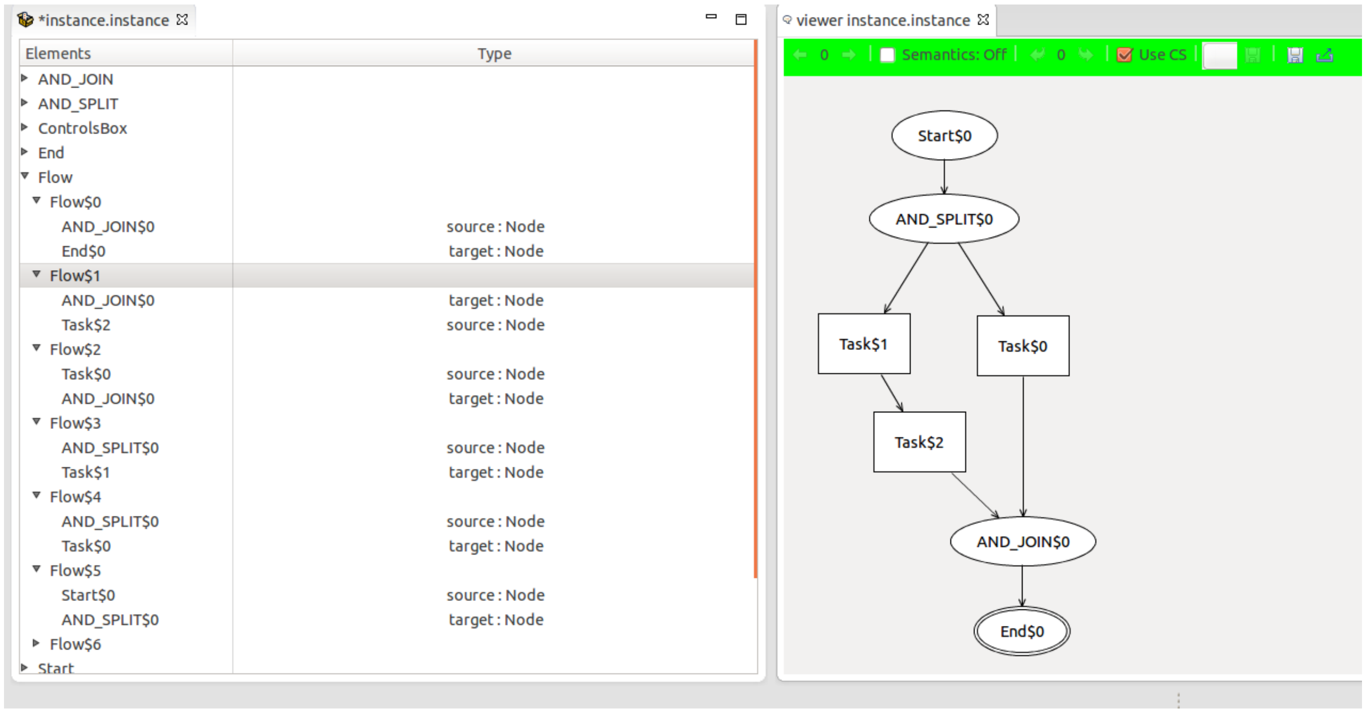


Fig. 5: Visualizing an instance using the concrete syntax


*instance.instance

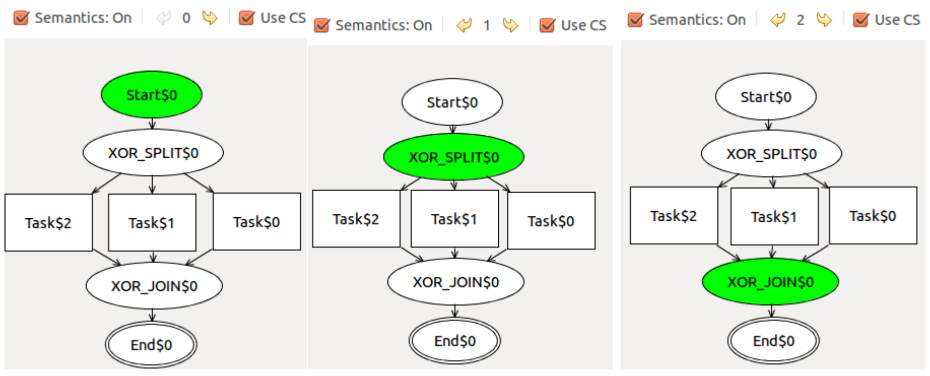
Elements	Type
AND_JOIN	
AND_SPLIT	
ControlsBox	
End	
Flow	
Flow$0	
Flow$1	
Flow$2	
Flow$3	
Flow$4	
Flow$5	
Start$0	source : Node
AND_SPLIT$0	target : Node
Start	
Task	
Task$0	
Task$1	
Task$2	

Fig. 6: Adding task to instance of fig.5 results in a non conforming instance



Elements	Type
▸ AND_JOIN	
▸ AND_SPLIT	
▸ ControlsBox	
▸ End	
▾ Flow	
▾ Flow$0	
AND_JOIN$0	source : Node
End$0	target : Node
▾ Flow$1	
AND_JOIN$0	target : Node
Task$2	source : Node
▾ Flow$2	
Task$0	source : Node
AND_JOIN$0	target : Node
▾ Flow$3	
AND_SPLIT$0	source : Node
Task$1	target : Node
▾ Flow$4	
AND_SPLIT$0	source : Node
Task$0	target : Node
▾ Flow$5	
Start$0	source : Node
AND_SPLIT$0	target : Node
▸ Flow$6	
▸ Start	

Fig. 7: Connecting this newly created task to the rest of the flow makes the instance conform to the ASM



(a) starting state (b) state 1 (c) state 2

Fig. 8: Faulty semantics execution of an SBP language model containing XORs