A Graphical Syntax for F-Alloy

Loïc Gammaitoni
Laboratory for Advanced Software Systems
University of Luxembourg
6, avenue de la Fonte
L-4364 Esch-sur-Alzette
Luxembourg

TR-LASSY-17-02

Abstract

F-Alloy is a formal language based on Alloy, aiming at defining transformations analyzable using the Alloy Analyser and computable in polynomial time using its dedicated tool, the F-Alloy interpreter. Whilst the presence of an interpreter greatly increases the usability of the language, one might still shun using F-Alloy due to the complexity of its formal syntax. In this paper, we propose a solution to this problem by enriching F-Alloy with a user-friendly graphical concrete syntax.

1 Motivation & Related Work:

There is evidence [5] that formal methods are proficient in improving the quality of one's design, yet designers of systems where correctness is not an absolute necessity (e.g. non safety-critical systems) are often reluctant to use them due to the complexity of their syntaxes.

Formalisms which are endowed with an alternative graphical syntax allow their users to view their designs from a different and broader angle – i.e., they provide an overview of the specification – hence facilitating their comprehension. This holds, e.g., for the widely used OCL language with the visual syntaxes proposed in [7] and [6]. For the case of model transformations, which are key artifacts in model driven engineering, a lot of effort is spent in making specifications more intuitive by using visual representations. Some model transformation formalisms are released with a visual syntax solely, like Henshin [1], while others who were initially introduced with a textual syntax, tend to later on offer an alternative visual syntax to their user; for instance, umlx [10] is a graphical language originating from an attempt to provide a graphical syntax to the textual model transformation language QVT-r [9].

In previous work [4] we introduced F-Alloy, a formalism used to specify model transformations in an agile manner – i.e., verification and validation of one's design can be done seamlessly at any stage of the design process [3]. One of the limitations of this work is the prerequisite of being familiar with the formal syntax of the language before being able to use it. As an attempt to tackle this limitation, we introduce in this paper a graphical concrete syntax to the F-Alloy language.

The rest of this paper is structured as follows. In the next section we introduce a model transformation from structured business processes to Petri nets, on which we rely throughout the paper to illustrate the use of F-Alloy (in section 3) and of its new graphical syntax (in section 4). We then validate our graphical syntax proposal by checking if it complies to the Physics of Notations proposed in [8] before closing the paper with concluding remarks and suggestions of future works.

2 The SBP2Pnet Transformation:

In this paper, we use the specification of the SBP2Pnet model transformation, a translation from a Structured Business Process language (SBP) to a Petri net language (Pnet), in order to illustrate the use of F-Alloy and of the graphical syntax we present in this paper. The source language of this transformation is SBP (Alloy model given[1] in listing 1). It contains the notion *nodes* and *flows*. Nodes can be *tasks* representing actions performed towards the completion of the process or *control nodes* structuring the process. They can also be *start* or *end* nodes marking the beginning and the end of the process. All nodes are

[1] Well-formedness constraints are left out for conciseness' sake. Full example can be found at http://lightning.gforge.uni.lu/examples/SBP2Pnet.zip

```
module SBP2Pnet/SBP

abstract sig Node{
  name: String
}
one sig Start,End extends Node {}
sig Task extends Node {}
sig Flow{
  source: Node,
  target: Node
}
abstract sig Control extends Node {}
sig AND_JOIN, XOR_JOIN, AND_SPLIT, XOR_SPLIT extends Control {}
```

Listing 1: specification of SBP in Alloy (no well-formedness constraints)

```
module SBP2Pnet/PetriNet

abstract sig Vertex{
  nextVertex: set Vertex,
  label: String
}
sig Place extends Vertex{}{
  nextVertex & Place = none
}
sig Transition extends Vertex{}{
  nextVertex & Transition=none
}
```

Listing 2: specification of Pnet in Alloy (no well-formedness constraints)

interconnected by *flows*. Control nodes can be of type AND or XOR, with the property that branches emitted by such nodes, should unite to a control node of same type. The source of the ramification is called a SPLIT node while the end is called a JOIN node. The operational semantics of SBP is as follows:

1. in the initial state the start node is active
2. in state n, the active nodes are the successors of nodes active in state n-1 except for XOR SPLIT nodes for which only one arbitrarily chosen successor is active, and AND JOIN nodes which should stay active until incoming ramifications do not contain any active nodes.

The target language of this transformation is Pnet (Alloy model given in listing 2). It contains the notion of *places*, *transitions* and *arcs* from places to

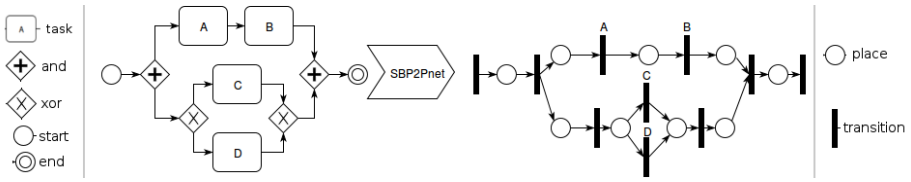

Figure 1: An SBP-model and its semantically equivalent Pnet-model as defined by the SBP2Pnet transformation

transitions and from transitions to places (modeled by the `nextVertex` relation). Places can contain tokens. The operational semantics of Pnet is as follows :

1. in the initial state, all places do not contain tokens
2. a transition whose incoming places have at least one token is said *enabled* and can be fired, resulting in the consumption of one token in each incoming place and the creation of one token in each outgoing place

The SBP2Pnet transformation produces for a given SBP-model (a model expressed in the SBP language) the semantically equivalent Pnet-model, knowing that SBP's active states are semantically equivalent to Pnet's enabled transitions. The application of the SBP2Pnet transformation for a given SBP-model is depicted in fig.1 while the F-Alloy code implementing the transformation is given in listing 3.

3 The F-Alloy Language

The F-Alloy language, as introduced in [4], allows the definition of exogenous transformations: model transformations for which input and output models are expressed in different languages. To specify such transformations, F-Alloy relies on the relational nature of Alloy, i.e., each transformation rule is embodied by a relation from a source element to its corresponding target element. We refer to those relations as mappings. The presence of a source element in one of such relation is conditioned by a *guard predicate*.

Properties of a target element are constrained in function of its corresponding source element in a `value predicate`.

To better understand the use of the language, we offer a reading of the SBP2Pnet transformation given in listing 3.

The SBP2PNet Transformation, as specified in listing 3, is composed of 4 mappings.

1. `node2Transition` creates for each node n – as the associated guard is empty – a transition t such that the label of t is the same than the name of n.

2. `flow2Place` creates a place for each flow whose source is not an `XOR SPLIT` and whose target is not an `XOR JOIN` (as, in those special cases, only a single place needs to be created and not one per flow as depicted in fig.1), so that the Place is succeeding the transition representing[2] the source of the flow, and is preceded by the transition representing the target of the flow.

3. `XOR_SPLIT2PLACE` creates a place for each XOR split. This created place is succeeding the transition representing the XOR split and is preceded by the transition representing the target of the flow whose source is the XOR split for which the place is created.

4. `XOR_JOIN2PLACE` creates a place for each XOR join. This created place is succeeding the transition representing the source of the flow whose target is the XOR join for which the place is created and is preceded by the transition representing this XOR join.

We can see that this transformation, though relatively simple, is already non trivial to process. In the next section we introduce a graphical syntax for F-Alloy which will hopefully increase the comprehension of specifications written in the language.

4 A Graphical Syntax for F-Alloy

In this section we provide evidence that F-Alloy specifications can be expressed graphically by associating each F-Alloy construct with a graphical representation and by using the obtained notations to depict, in fig.2 the SBP2Pnet transformation, as defined in listing 3.

F-Alloy specifications are represented by big boxes, whose header gives information on the source and target model of the transformation. Those boxes contain blocks representing the mappings declared in the F-Alloy specification, each block being labeled after the name of the mapping they represent. Those block contains all the information the mapping and its associated predicates carry, that is: (1) the nature of elements that are related by this mapping (2) the condition under which elements are part of this mapping (3) the properties of elements in the range of the mapping expressed as a function of elements in the domain. Those pieces of information are represented as follows:

1. Mappings are represented by thick red arrows. At the root and head of those arrows are the source and target element of mappings, respectively. All those elements are central to the transformation definition, and are thus highlighted.

2. The representation of conditions under which elements are part of a mapping is composed of:

[2] as defined by the `node2Transition` mapping

- *Navigation arrows*: plain black arrows labeled after a property of the element it originates from – e.g., in the `flow2Place` block of fig.2, the arrow labeled "source" points to the node being the source of the flow it originates from.
- *Categories*: A double-lined rectangle representing the set of all elements conforming to its type – e.g., in the `flow2Place` block, the double-lined rectangle labeled: "`XOR_SPLIT`" represents the set of all XOR split elements in the input of the transformation.
- *Set membership arrows*: black arrows with rounded head, they represent the condition that the element at its source should be part of the set it points too. This condition can be negated by placing a red cross over the arrow – e.g., in the `flow2Place` block, it is specified that for a flow to be part of the mapping, its source and target should not belong to the set of `XOR_SPLITs` and `XOR_JOINs`, respectively.
- *Sets*: depicted by two boxes, one on top of the other, they represent the fact that multiple element might be at the source or target of a navigation arrow due to the multiplicity of the property this latter is associated to – e.g., in the `XOR_SPLIT2Place` block, as an `XOR_SPLIT` can be the source of multiple flow, the set notation is used to represent those flows which are source of the said `XOR_SPLIT`.

3. The properties of elements in the range of mappings are defined by using:

   - *References* to the output of other mappings: denoted by an arrow representing a mapping, whose target is a box representing the image of the element at its source. Both the arrow and its target are depicted in red, this color being used exclusively for depicting elements directly related to mappings – e.g, in the `flow2Place` block, we refer to the transition t1, as being the image of Flow's source (:Node) via the `node2Transition` mapping.
   - *Navigation arrows* pointing at the values the property should take – e.g., in the `flow2Place` block, for any flow f mapped to a place p, the next vertex of p is the transition which has been associated to the target of f via the `node2Transition` mapping.
   - *Partial assignment arrows* ($\overset{\circ}{\leftarrow}$): the set of values of the property after which the arrow is named is not limited to the element the arrow is pointing to (other mappings might contribute to the valuation of this property).
   - *Direct assignment*: when properties of the input and output elements have the same value, the properties appear inside the boxes representing those elements similarly to attributes in an object diagram's notation, that is, following the format "propertyName = **x**", where **x** is a variable referring to the value of the property. Two properties have thus the same value if the same variable is assigned to them –

e.g, in the `node2Transition` block, transitions are labeled after the name of the node they represent.

In the next section, we review the proposed notation in the light of the nine principles detailed in Moody's physics of notation [8].

5 Validation

In this section, we see how the nine principles enumerated in Moody's Physics of Notation [8] are respected by the syntax we proposed.

- *Semiotic Clarity:* This principle has been followed when designing the notations used to depict mappings, conditions, and value assignments, but has been neglected for the representation of guard and value predicates, which do not appear as two distinct entities in the graphical notation we propose. This decision was taken to avoid unnecessary redundancies, as elements present in a guard predicate ought to reappear in the accompanying value predicate.
- *Perceptual Discriminability:* Elements of the representation are easily differentiable from the background and semantically non-equivalent elements
- *Semantic transparency:* has been achieved by reusing notations present in object diagrams to represent elements and relations between those.
- *Complexity Management:* The complexity of comprehending the full transformation is reduced to the comprehension of each of the chunk it is composed of, i.e., the blocks representing mappings.
- *Cognitive Integration:* The name of concepts declared in the transformation's source and target meta-model is always present in the label of elements they type, hence allowing users to relate.
- *Visual Expressiveness:* could be improved by adding texture and additional shapes to the notation. Yet the number of different construct composing the notation being low, further enhancing visual expressiveness seems unnecessary.
- *Dual Coding:* We extensively use text to complement our graphical notation.
- *Graphic Economy:* The number of different graphical symbol in use is minimal hence remaining cognitively manageable.
- *Cognitive fit:* We used an object-diagram styled notation for an audience which is essentially composed of language designers and modelers. This audience should be familiar with this kind of notation.

As shown, most of those nine principles were considered when designing the proposed notation, which gives us a first validation of our syntax's design choice. Of course, further validation, based on empirical studies are needed to confirm the relevance of the graphical syntax suggested in this paper.

6 Conclusion and Future Work

This paper gives a quick overview of an ongoing work: extending F-Alloy's usability by complementing the language with a graphical syntax. Future works includes the formal definition of this syntax as a function of F-Alloy's abstract syntax, as well as the implementation of this definition in the Lightning tool[2], a language workbench based on Alloy. Also, an empirical study is planned to determinate whether or not this notation improves the readability and comprehension of F-Alloy specifications and hence further validate this notation proposal.

```
module SPBP2Pnet/SBP2Pnet

open SBP2Pnet/SBP
open SPB2Pnet/PetriNet

one sig Create{
 node2Transition: Node->Transition,
 flow2Place: Flow->Place,
 XOR_SPLIT2Place: XOR_SPLIT -> Place,
 XOR_JOIN2Place: XOR_JOIN -> Place
}
pred guard_node2Transition(n:Node) {}
pred value_node2Transition (n:Node, t:Transition) {
   t.label= n.name
}
pred guard_flow2Place(f:Flow) {
 f.source not in XOR_SPLIT and f.target not in XOR_JOIN
}
pred value_flow2Place(f:Flow, p:Place) {
 p in Bridge.node2Transition
     [f.source].nextVertex
 p.nextVertex= Bridge.node2Transition
     [f.target]
}
pred guard_XOR_SPLIT2Place(n:XOR_SPLIT) {}
pred value_XOR_SPLIT2Place(n:XOR_SPLIT, p:Place) {
 p in Bridge.node2Transition[n].nextVertex
 p.nextVertex=
     Bridge.node2Transition[n.~source.target]
}
pred guard_XOR_JOIN2Place(n:XOR_JOIN) {}
pred value_XOR_JOIN2Place(n:XOR_JOIN, p:Place) {
 p in Bridge.node2Transition
     [n.~target.source].nextVertex
 p.nextVertex= Bridge.node2Transition[n]
}
```

Listing 3: F-Alloy specification of the SBP2Pnet model Transformation

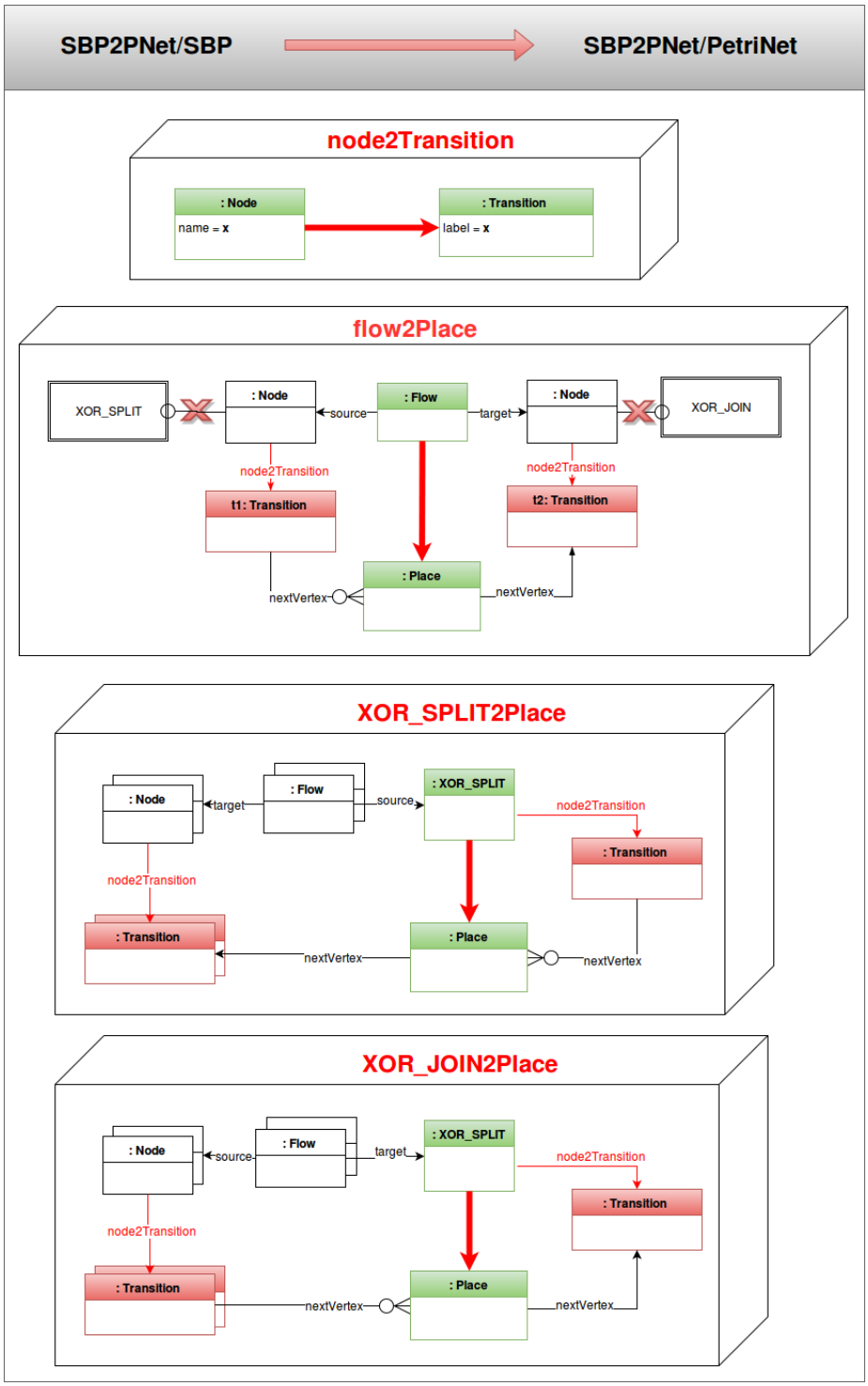

Figure 2: The SBP2Pnet transformation defined in listing 3 visualized using the proposed graphical syntax

References

[1] Ermel and al. Visual modeling of controlled emf model transformation using henshin. *Electronic Communications of the EASST*, 2011.

[2] Loïc Gammaitoni, Pierre Kelsen, and Christian Glodt. Designing languages using lightning. In *International Conference on Software Language Engineering*, 2015.

[3] Loïc Gammaitoni, Pierre Kelsen, and Fabien Mathey. 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)*, pages 19–28. 2014.

[4] Loïc Gammaitoni and Pierre Kelsen. F-Alloy: an Alloy Based Model Transformation Language. In *International Conference on Model Transformations*. 2015.

[5] Daniel Jackson. *Software abstractions*. MIT Press Cambridge, 2012.

[6] Stuart Kent. Constraint diagrams: visualizing invariants in object-oriented models. In *ACM SIGPLAN Notices*, volume 32, pages 327–341. ACM, 1997.

[7] Chr Kiesner, Gabriele Taentzer, and Jessica Winkelmann. Visual ocl: A visual notation of the object constraint language. *http://tfs.cs.tu-berlin.de/vocl*, 2002.

[8] Daniel L Moody. The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. *Software Engineering, IEEE Transactions on*, 35(6):756–779, 2009.

[9] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1. 2011.

[10] Edward D Willink. A concrete uml-based graphical transformation syntax-the uml to rdbms example in umlx. In *Workshop on Metamodelling for MDA*, 2003.