A Lightweight Approach for Defining the Formal Semantics of a Modeling Language

Pierre Kelsen and Qin Ma

Laboratory for Advanced Software Systems
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg
{Pierre.Kelsen, Qin.Ma}@uni.lu

Abstract. To define the formal semantics of a modeling language, one normally starts from the abstract syntax and then defines the static semantics and dynamic semantics. Having a formal semantics is important for reasoning about the language but also for building tools for the language. In this paper we propose a novel approach for this task based on the Alloy language. With the help of a concrete example language, we contrast this approach with traditional methods based on formal languages, type checking, meta-modeling and operational semantics. Although both Alloy and traditional techniques yield a formal semantics of the language, the Alloy-based approach has two key advantages: a uniform notation, and immediate automatic analyzability using the Alloy analyzer. Together with the simplicity of Alloy, our approach offers the prospect of making formal definitions easier, hopefully paving the way for a wider adoption of formal techniques in the definition of modeling languages.

1 Introduction

To fully define a modeling language one needs to specify: (1) the abstract syntax (the structure of its models), (2) the concrete syntax (the actual notation presented to the user), (3) the static semantics (well-formedness rules for the models), (4) the dynamic semantics (the meaning of a model). Ideally all these elements of a modeling language are formally defined. By having a formal definition one can precisely reason about the language; by eliminating any ambiguities inherent to informal descriptions such a formal specification is also a good starting point for developing tool support.

Despite the widely recognized advantages of a formal description, modeling languages are often introduced without a full formal definition. A good example is the Unified Modeling Language [11]. This modeling language is the de facto standard modeling language in software engineering. Nevertheless the semantics of the UML is not fully formally defined. This complicates the development of UML tools and compromises their interoperability [1].

This paper focuses on the formal definition of the semantics. The term "formal semantics" in the title usually refers to the dynamic semantics only (see, e.g., [13]). Because the dynamic semantics intimately relates to the abstract syntax and the static semantics but is independent of the concrete syntax, we study in this paper approaches for defining abstract syntax, static semantics and dynamic semantics using a concrete example language. We first present two traditional approaches for defining abstract syntax and static semantics based on EBNF and type systems on one hand, and metamodeling on the other hand. For dynamic semantics we present one traditional approach based on operational semantics. We then propose a uniform approach for defining these aspects of a modeling language based on Alloy ([4]). Besides having a single language for this task, Alloy provides the additional benefit of providing automatic tool support for checking the correctness of semantic specifications. Automatic analysis facilitates incremental development of complex semantic descriptions by uncovering errors early. The term "lightweight" used in the title of the paper was chosen because Alloy attempts to obtain the benefits of traditional formal methods at lower cost [5].

We now discuss related work. We illustrate the Alloy-based approach for defining the formal semantics by looking at a concrete language which is a subset of the EP language, a declarative executable modeling language [6, 8]. The same exercise was done for the full EP language and similar conclusions were reached ([7]). We only present one traditional approach for specifying dynamic semantics: operational semantics. For a fuller account of this approach the reader is referred to [13]. Other approaches for specifying the dynamic semantics are [10, 14]: denotational semantics (map models to denotations and reason in terms of the denotations), axiomatic semantics (exploiting assertions about programs), and hybrid approaches.

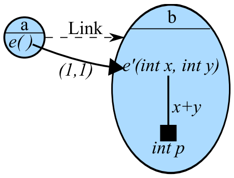
Although much effort has been dedicated to establishing firm theoretical foundations for formal semantics during the last four decades, it is recognized that there is still no universally accepted formal notation for semantic description [14]. More importantly formal semantics approaches currently have limited practical use. Possible hindrances cited by Mosses (e.g., [9]) are readability and lack of tool support, which we will address with the Alloy approach.

This rest of the paper is structured as follows: in the next section we present the concrete modeling language that will be used as a running example for the remainder of the paper. In section 3 we describe traditional approaches for defining abstract syntax and static and dynamic semantics of the example language. In section 4 we do the same exercise using Alloy. We then compare in section 5 the Alloy-based approach with the traditional approaches. We close with concluding remarks in the final section.

2 Example Language

The chosen EP subset is specified by the metamodel in Figure 1. An EP *system* defines a set of *models*, each of which consists of a set of *properties* and a set of *events*. EP models play a role similar to classes in an object-oriented system. *Instances* of models exist at runtime, where the value assignment to their properties constitutes the state of the system, and events represent available operations on these instances. For simplicity, we omit primitive types, i.e., types are models and values are instances. Each valid system should have exactly one main model, considered as the entry point of a system, in the sense that the initial state of an EP system is acquired by creating an unique instance of it.

The novelty of the EP language is that behavior of events is modeled declaratively via *event edges*. Briefly, execution of an event on an instance can influence the state of a system in two ways: it can either change the value of a property of the instance using an *impact edge*, where the new value is evaluated from the expression associated with this edge; or trigger other events on some instances via *push edges*, which then proceed in a similar fashion. A push edge is a directional connection from a source event to a target event. It has a link property whose value indicates on which instance the target event will be triggered. Because each event can have *parameters*, the push edge specifies for each parameter in the target event an expression that computes the value of this parameter. (We omit details of the expression language in this paper because of the limited space; in [8] we showed that OCL is a good candidate language.)

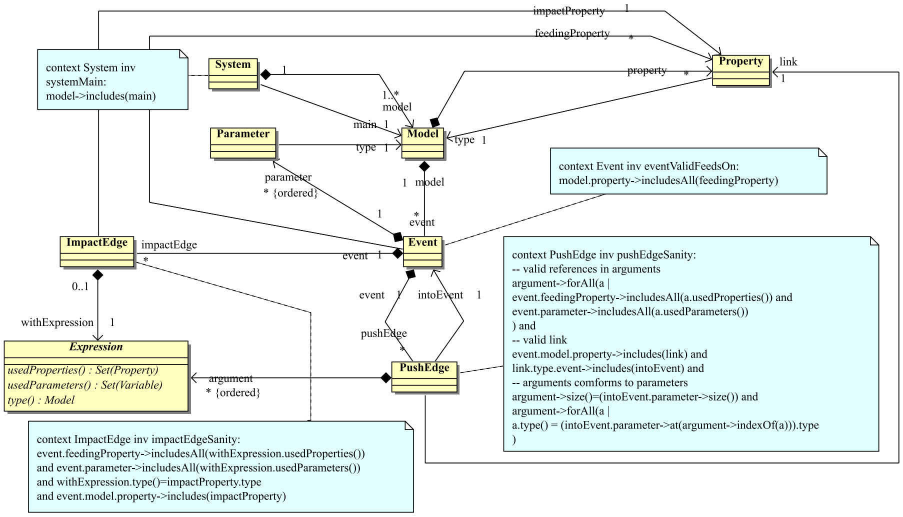


We close our informal description of the EP subset by looking at an example of two instances “a” and “b” existing at runtime and belonging to some models in which events and event edges are defined as in the graph to the left. Instance “a” has an event “e” with no parameters, and instance “b” has an event “e'” that takes two integer parameters. The triangle-headed solid arrow denotes the push edge between events; the dashed arrow labeled “Link” specifies the link of the edge which is basically a local property of “a” that points to “b”. In comparison, the square-headed arrow between event “e'” and local property “p” denotes an impact edge, labeled by expression $x + y$, i.e., the sum of the two parameters of the event. In a nutshell, this example depicts the following event propagation and local property modification: (1) Execution of “e” on “a” would trigger execution of “e'” on “b” with the constant arguments 1 and 1; (2) this would in turn change the value of “p” of “b” to the sum of the two arguments of “e'”, i.e., 2.

3 Traditional Approaches

3.1 Abstract syntax and static semantics

3.1.1 EP metamodel and OCL constraints Figure 1 defines the abstract syntax of the chosen EP subset as a meta-model and the static semantics as OCL [12] constraints. Four groups of well-formedness constraints are specified, in addition to the inherent syntax restrictions designated by the class diagram, such as compositions and multiplicities. For any valid system, the appointed *main* model should be defined in the system, and for any valid event, the set of feeding properties should all be defined in the model that also owns the event. The sanity rules for impact and push edges in event definitions are a bit more



```
context System inv
systemMain:
model->includes(main)
```

```
context Event inv eventValidFeedsOn:
model.property->includesAll(feedingProperty)
```

```
context PushEdge inv pushEdgeSanity:
-- valid references in arguments
argument->forAll(a |
event.feedingProperty->includesAll(a.usedProperties()) and
event.parameter->includesAll(a.usedParameters())
) and
-- valid link
event.model.property->includes(link) and
link.type.event->includes(intoEvent) and
-- arguments comforms to parameters
argument->size()=(intoEvent.parameter->size()) and
argument->forAll(a |
a.type() = (intoEvent.parameter->at(argument->indexOf(a))).type
)
```

```
context ImpactEdge inv impactEdgeSanity:
event.feedingProperty->includesAll(withExpression.usedProperties())
and event.parameter->includesAll(withExpression.usedParameters())
and withExpression.type()=impactProperty.type
and event.model.property->includes(impactProperty)
```

Fig. 1. EP subset meta-model with constraints in OCL

involved. The following constraints must be satisfied in order for an impact edge to be valid: (1) the expression that specifies the new value of the impacted property should only reference properties that feed the event and event parameters; (2) the expression and the impacted property should have the same type; (3) the impacted property and the event that owns the edge should both be owned by the same model.

For a push edge to be valid, the following constraints are enforced: (1) the expressions used to compute arguments of the target event should only reference parameters of the source event or properties that feed the source event; (2) the link of the edge should point to a property, whose type, which is a model, owns the target event. (3) the sequence of arguments and the sequence of parameters of the target event should have the same size and the same type, respectively.

3.1.2 EBNF syntax and typing rules

Another way to formalize the abstract syntax and the static semantics of the example language is to follow the approach usually adopted in specifying formal languages, namely via EBNF based syntax and type checking rules.

The EBNF based abstract syntax appears on top of Figure 2. Different from the EP metamodel in which all syntax entities are referred to by direct references, this approach is textual, i.e., references are via names. As a consequence, we assume the following disjoint sets of names: variable names $x \in \mathcal{X}$, where **self** is a special variable denoting the recursive "self" reference; property names $p \in \mathcal{P}$; event names $e \in \mathcal{E}$; and model names $m \in \mathcal{M}$. Moreover, we write $\overline{\mathtt{XX}}$ for a sequence of form $\mathtt{XX}_1, \ldots, \mathtt{XX}_k$. An alternative notation is $\mathtt{XX}_i{}^{1 \leq i \leq k}$, or sim-

EBNF abstract syntax:

$$
\begin{array}{lll}
\mathtt{Sys} & ::= \overline{\mathtt{Mdf}} & \text{EP systems} \\[1em]
\mathtt{Mdf} & ::= \textsf{\textbf{model}}\ m\ \{\overline{m}\ \overline{p};\ \overline{\mathtt{Edf}}\} & \text{Model definitions} \\[1em]
\mathtt{Edf} & ::= \overline{p}\ \textsf{\textbf{feed}}\ e(\overline{m}\ \overline{x})\ \{\overline{\mathtt{Edg}}\} & \text{Event definitions} \\[1em]
\mathtt{Edg} & ::= \textsf{\textbf{push}}\ \overline{\mathtt{Exp}}\ \textsf{\textbf{into}}\ e\ \textsf{\textbf{via}}\ p & \text{Push edges} \\
 & \ \textsf{\textbf{impact}}\ p\ \textsf{\textbf{with}}\ \mathtt{Exp} & \text{Impact edges}
\end{array}
$$

Typing environments:

$$\mathit{\Gamma} ::= \epsilon \mid (x : m{\upharpoonright}\overline{p}), \mathit{\Gamma}$$

Type checking rules:

TSystem

$$\dfrac{\begin{array}{l}(\vdash \mathtt{Mdf}_i \ ::\ \textsf{\textbf{ok}})^{\,i\in I} \\ \textsf{Main is declared} \\ \overline{\mathtt{Mdf}_i}^{\,i\in I} \text{ declare distinct model names}\end{array}}{\vdash \overline{\mathtt{Mdf}_i}^{\,i\in I} \ ::\ \textsf{\textbf{ok}}}$$

TModel

$$\dfrac{\begin{array}{l}(\vdash^{(m,\overline{p})} \mathtt{Edf}_i \ ::\ \textsf{\textbf{ok}})^{\,i\in I} \\ \overline{p} \text{ distinct} \\ \overline{\mathtt{Edf}_i}^{\,i\in I} \text{ declare distinct names}\end{array}}{\vdash \textsf{\textbf{model}}\ m\ \{\overline{m}\ \overline{p}, \overline{\mathtt{Edf}_i}^{\,i\in I}\} \ ::\ \textsf{\textbf{ok}}}$$

TEvent

$$\dfrac{\overline{p} \subseteq \overline{q} \qquad \overline{x} \text{ distinct} \qquad ((\overline{x} : \overline{m}) \vdash^{(m,\overline{p})} \mathtt{Edg}_i \ ::\ \textsf{\textbf{ok}})^{\,i\in I}}{\vdash^{(m,\overline{q})} \overline{p}\ \textsf{\textbf{feed}}\ e(\overline{m}\ \overline{x})\ \{\overline{\mathtt{Edg}_i}^{\,i\in I}\} \ ::\ \textsf{\textbf{ok}}}$$

TPushEdge

$$\dfrac{\overline{\mathit{etype}(\mathit{ptype}(m,p),e) = m_i}^{\,i\in I} \qquad (\mathit{\Gamma} \cup (\textsf{\textbf{self}} : m{\upharpoonright}\overline{p}) \vdash \mathtt{Exp}_i \ ::\ m_i)^{\,i\in I}}{\mathit{\Gamma} \vdash^{(m,\overline{p})} \textsf{\textbf{push}}\ \overline{\mathtt{Exp}_i}^{\,i\in I}\ \textsf{\textbf{into}}\ e\ \textsf{\textbf{via}}\ p \ ::\ \textsf{\textbf{ok}}}$$

TImpactEdge

$$\dfrac{\mathit{\Gamma} \cup (\textsf{\textbf{self}} : m{\upharpoonright}\overline{p}) \vdash \mathtt{Exp} \ ::\ \mathit{ptype}(m,p)}{\mathit{\Gamma} \vdash^{(m,\overline{p})} \textsf{\textbf{impact}}\ p\ \textsf{\textbf{with}}\ \mathtt{Exp} \ ::\ \textsf{\textbf{ok}}}$$

Auxiliary definitions:

EventTypeLookUp

$$\dfrac{\begin{array}{c}\textsf{\textbf{model}}\ m\ \{\overline{m_1}\ \overline{p_1}; \overline{\mathtt{Edf}}\} \in \mathtt{Sys} \\ \overline{p_2}\ \textsf{\textbf{feed}}\ e(\overline{m}\ \overline{x})\ \{\overline{\mathtt{Edg}}\} \in \overline{\mathtt{Edf}}\end{array}}{\mathit{etype}(m,e) = \overline{m}}$$

PropertyTypeLookUp

$$\dfrac{\textsf{\textbf{model}}\ m\ \{\overline{m_1}\ \overline{p_1}, m'\ p', \overline{m_2}\ \overline{p_2}; \overline{\mathtt{Edf}}\} \in \mathtt{Sys} \qquad p' \in \overline{p}}{\mathit{ptype}(m{\upharpoonright}\overline{p}, p') = m'}$$

Fig. 2. EBNF abstract syntax, typing, and auxiliary functions

ply $\mathtt{XX}_i{}^{i \in I}$ where I stands for the set of possible subscripts. We abbreviate a sequence of pairs as a pair of sequences, namely writing $\overline{\mathtt{XX}}\textvisiblespace\overline{\mathtt{YY}}$ as shorthand for $\mathtt{XX}_1\textvisiblespace\mathtt{YY}_1, \ldots, \mathtt{XX}_k\textvisiblespace\mathtt{YY}_k$. An empty sequence is denoted by ϵ.

Typing rules appear in the middle of Figure 2. The special type **ok** denotes well-typedness. The typing environment Γ is a finite binding from variables to types, in which the mask operation on models: $m{\restriction}\overline{p}$, allows us to control the visibility of properties of model m in different circumstances. Putting a mask $\overline{p}$ on m restricts only those properties of m appearing in $\overline{p}$ to be visible. A default mask is $\mathcal{P}$, which is usually omitted; it allows all properties to be visible.

Two auxiliary functions are used in typing rules, defined in the bottom of Figure 2. Function $etype(m, e)$ returns the sequence of types of the parameters of event e as defined in model m, or otherwise returns $\perp$, denoting an error during type checking. Function $ptype(m{\restriction}\overline{p}, p)$ returns the type of property p as defined in model m if not masked (i.e. $p \in \overline{p}$), or otherwise $\perp$.

Typing rules in principle correspond to the OCL constraints discussed in Section 3.1.1. For example, the first premise in rule TEvent corresponds to the eventValidFeedsOn invariant of Event (see Figure 1). Moreover, the enforcement of the three constraints discussed on page 4 for impact edges is distributed across typing rules as follows: (1) the typing environment in which the expression of the impact edge is typed consists of two parts: the first part is the event parameters accumulated in the second premise of rule TEvent, and the second part is the set of properties that feed the event, accumulated in the premise of rule TImpactEdge. In other words, the only possible references from the expression fall into these two categories; (2) the type of the expression is required to be the same as the type of the impacted property p, as depicted by the premise of rule TImpactEdge; (3) the ownership of model m to p is assured when computing $ptype(m, p)$, because otherwise $\perp$ will be returned.

Similarly, the sanity constraints of PushEdge are all spelled out, too, across the premises of rules TEvent and TPushEdge.

However, because of the textual nature of the approach, additional bookkeeping and name distinction premises are required, such as in rule TSystem and TModel. But we also gain from the naming facility. For example, we spare the systemMain invariant of System, because asking for a unique model defined in the system with name Main suffices to assure its ownership thanks to name scoping: in the scope of the system, name Main will always refers to the unique model in the system with name Main.

3.2 Operational semantics

In this section we assume that the reader is familiar with operational semantics [13]. A system configuration (Λ, s) is a pair of an evaluation environment and a state. As defined on top of Figure 3, the state records the current set of instances, and the environment, binding variables to values, tells how to access these instances via variables. A value $v \in \mathcal{V}$ is either **null** denoting void, or an identity $id \in \mathcal{ID}$ denoting an instance of a certain model. Instance identities are distinct. In the state, an instance identity is bound with a pair (m, φ_m). The

Values, states, and substitutions:

$$\begin{array}{lll}
\text{States:} & s & ::= \emptyset \mid (id : (m, \varphi_m)), s \\
\text{Values:} & v & ::= \textsf{\textbf{null}} \mid id \\
\text{Evaluations:} & \varphi_m & : \quad \mathcal{P} \to \mathcal{V} \\
\text{State substitutions:} & \sigma & : \quad \mathcal{ID} \times \mathcal{P} \to \mathcal{V} \\
\text{Substitution application:} & &
\end{array}$$

$$\begin{aligned}
s\sigma(id).m &= s(id).m \\
s\sigma(id).\varphi_m(p) &= \begin{cases} s(id).\varphi_m(p) & \sigma(id, p) = \bot \\ \sigma(id, p) & \text{otherwise} \end{cases}
\end{aligned}$$

Evaluation environments:

$$\varLambda ::= \epsilon \mid (x : v), \varLambda$$

Expression evaluation (partial):

$$\textsc{Variable}\quad \frac{\varLambda(x) = v}{\varLambda \vDash (x,\ s) \Downarrow (v,\ \emptyset)} \qquad \textsc{InstanceCreation}\quad \frac{id \text{ fresh}}{\varLambda \vDash (m :: \textsf{\textbf{create}}(),\ s) \Downarrow (id,\ (id : (m, \phi_m)))}$$

$$\textsc{PropertyCall}\quad \frac{\varLambda \vDash (\texttt{Exp},\ s) \Downarrow (id,\ s_\triangle) \qquad (s \cup s_\triangle)(id).\varphi_m(p) = v}{\varLambda \vDash (\texttt{Exp}.p,\ s) \Downarrow (v,\ s_\triangle)}$$

Edge evaluation:

$$\textsc{EImpactEdge}\quad \frac{\varLambda \vDash (\texttt{Exp},\ s) \Downarrow (v_1,\ s_\triangle^1) \qquad \varLambda \vDash (\textsf{\textbf{self}},\ s) \Downarrow (v_2,\ s_\triangle^2)}{\varLambda \vDash (\textsf{\textbf{impact}}\ p\ \textsf{\textbf{with}}\ \texttt{Exp},\ s) \Downarrow (((v_2, p) : v_1),\ s_\triangle^1 \cup s_\triangle^2)}$$

$$\textsc{EPushEdge}\quad \frac{\varLambda \vDash ((\textsf{\textbf{self}}.p).e(\overline{\texttt{Exp}}),\ s) \Downarrow (\sigma,\ s_\triangle)}{\varLambda \vDash (\textsf{\textbf{push}}\ \overline{\texttt{Exp}}\ \textsf{\textbf{into}}\ e\ \textsf{\textbf{via}}\ p,\ s) \Downarrow (\sigma,\ s_\triangle)}$$

Event call evaluation:

$$\textsc{EventCall}\quad \frac{\begin{array}{ll} \varLambda \vDash (\texttt{Exp},\ s) \Downarrow (v,\ s_\triangle) & v \neq \textsf{\textbf{null}} \\ \varLambda \vDash (\overline{\texttt{Exp}},\ s) \Downarrow (\overline{v},\ \overline{s_\triangle}) & \overline{p}\ \textsf{\textbf{feed}}\ e(\overline{m}\ \overline{x})\ \{\texttt{Edg}_i{}^{i \in I}\} \in (s \cup s_\triangle)(v).m \\ \multicolumn{2}{l}{(\varLambda \cup (\overline{x} : \overline{v}) \cup (\textsf{\textbf{self}} : v) \vDash (\texttt{Edg}_i,\ s \cup s_\triangle \cup \overline{s_\triangle}) \Downarrow (\sigma_i,\ s_\triangle^i))^{i \in I}} \end{array}}{\varLambda \vDash (\texttt{Exp}.e(\overline{\texttt{Exp}}),\ s) \Downarrow (\biguplus_{i \in I} \sigma_i,\ s_\triangle \cup \overline{s_\triangle} \cup (\bigcup_{i \in I} s_\triangle^i))}$$

Event invocation reduction:

$$\textsc{EventInvocationReduction}\quad \frac{\varLambda \vDash (\texttt{Exp}.e(\overline{\texttt{Exp}}),\ s) \Downarrow (\sigma,\ s_\triangle)}{(\varLambda,\ s) \xrightarrow{\texttt{Exp}.e(\overline{\texttt{Exp}})} (\varLambda,\ (s \cup s_\triangle)\sigma)}$$

Fig. 3. Operational semantics

former denotes the model of the instance and the latter gives its evaluation. An evaluation of an instance of model m, written φ_m, is a partial function from properties to values where $\forall p \in \mathcal{P}$, $\varphi_m(p) = \bot$ (i.e. undefined) if and only if p is not a property of m. For each model m, we assume a default evaluation for its instances, denoted by ϕ_m, in which we associate **null** to all properties defined in m. Given a state s and an instance id in it, we use $s(id).m$ to denote the model of the instance, and $s(id).\varphi_m$ the corresponding evaluation. Finally, system state substitution, denoted by σ, is a partial function from pairs of instance identities and properties to values. For a given substitution σ, $\sigma(id, p) = v$ means to change the value of p of instance id into v, and $\sigma(id, p) = \bot$ means to leave this value unchanged. As a consequence, applying the substitution to a state s, written $s\sigma$, returns a new state whose definition is also presented in Figure 3. Sometimes (for instance in rule EImpactEdge of Figure 3), we specify a substitution by directly listing the triples defined in it as: $\sigma ::= \emptyset \mid ((id, p) : v), \sigma$.

Upon initialization, the system configuration, (Λ_0, s_0), has a unique instance in s_0 of identity id where $s_0(id).m = \mathsf{Main}$ and $s_0(id).\varphi_m = \phi_{\mathsf{Main}}$ (the default evaluation of the main model), and a unique binding $(\mathsf{main} : id)$ in Λ_0. Interacting with an EP system amounts to invoking some event on some instance accessible from variable main and changing the system into a new configuration. The formal reduction relation is given at the bottom of Figure 3. Note that the evaluation environment Λ does not change after the reduction, because invoking an event as by $\texttt{Exp}.e(\overline{\texttt{Exp}})$ would not change the set of global variables nor their bindings. However, the set of accessible instances from variable main may still evolve, for instance by changing the value of the properties of the main instance.

Let us elaborate on the evaluation rules for event calls and edges. The judgment $\Lambda \vDash (\texttt{Exp}.e(\overline{\texttt{Exp}}),\, s) \Downarrow (\sigma,\, s_\Delta)$ means: calling event e under environment Λ and in state s evaluates to a substitution σ and a set of new instances s_Δ created during the evaluation. In rule EventCall we first evaluate the callee Exp into a value v [1], which should not be **null** in order for the calling to be meaningful, hence is an instance identity. Then, the arguments $\texttt{Exp}_i{}^{i \in I}$ are evaluated respectively. We pick up the body of the called event from the definition of $(s \cup s_\Delta)(v).m$, which effectively denotes the model of the callee, and evaluate the body, i.e. the edges, in a new evaluation environment extended with the parameters all bound to the just computed argument values, and **self** bound to the callee instance.

Evaluation judgments for edges take the form: $\Lambda \vDash (\texttt{Edg},\, s) \Downarrow (\sigma,\, s_\Delta)$. In rule EImpactEdge evaluating an impact edge would result in a substitution of form $((v_2, p) : v_1)$, which basically asks to change the value of the impacted property p on instance denoted by **self** (i.e. v_2) into the new value (i.e. v_1) computed from the expression Exp. By contrast, the result of evaluating a push edge (rule EPushEdge) is derived from recursively calling the target event on the instance denoted by the link property p of **self** within the same evaluation environment and state.

[1] Being out of the scope of the paper, expression evaluation rules are only partially presented in Figure 3 to help readers understand the other parts.

The second part in the results of evaluations, namely s_Δ, keeps track of the new instances that are created during the evaluations. As new instances can only be created in **create** expressions (rule InstanceCreation), where we have already required the freshness of identity, the accumulated s_Δ's would never conflict on identities, neither with each other nor with an instance in the original state. Therefore, the plain set union operation suffices. By contrast, in order to keep the dynamic semantics deterministic, namely at most one modification of a property of an instance is called for during an event invocation, we use the disjoint set union, denoted by $\uplus$, when collecting the substitutions along the evaluation, to enforce disjoint domains.

Finally, in rule EventInvocationReduction, the accumulated substitutions and new instances take effect by applying to the original state and reach a new state: $(s \cup s_\Delta)\sigma$.

4 The Alloy Approach

4.1 Overview of Alloy

Alloy [4] is a language that expresses software abstractions precisely and succinctly. A system is modeled in Alloy using a set of types called *signatures*. Each signature may have a number of *fields*. Constraints may be added as *facts* to a system to express additional properties. In terms of rigor Alloy rivals traditional formal methods. Its novelty is the *Alloy Analyzer*, a tool that allows fully automatic analysis of a system; it can expose flaws early and thus encourages incremental development. Two types of analyses can be performed using the Alloy Analyzer: we can search for an instance (obtained by populating signatures with elements) satisfying a predicate and we can look for a counterexample for an assertion; both assertions and predicates are part of the Alloy model. Both types of analyses rely on the *small scope hypothesis* ([4]): only a finite subspace is searched based on the assumption that if there is an instance or a counterexample there is one of small size.

In the remainder of this section we present the Alloy models representing the syntax and semantics of the example language. To fully understand the models, the reader needs to be familiar with the Alloy language. We will, however, comment the models so that readers new to Alloy should still be able to understand the salient features of this approach.

4.2 Abstract Syntax and Static Semantics

The Alloy model given below expresses both abstract syntax and static semantics. We will explain it by comparing it to the metamodel of figure 1. Each of the eight classes in the metamodel has a corresponding signature with a similar name in the Alloy model. For instance, the *System* class is represented by the *EPSystem* signature. Associations in the metamodel are usually represented by fields of a signature: thus the field *models* in *EPSystem* corresponds to the aggregation from the *System* class to the *Model* class in the metamodel. Multiplicities on the association ends in the metamodel are normally expressed as constraints in the Alloy model. For instance, the fact that each *Model* is contained in a single *System* is expressed as the signature fact (i.e., a fact associated with each element of a signature) `{one s:EPSystem | this in s.models}` in signature *EPModel*. There are two signatures in the Alloy model that do not correspond to classes of the metamodel: the *Main* signature expresses the fact that there is a single distinguished model called *Main*: this is expressed in the metamodel by an association from *System* to *Model*. The signature *ParameterMapping* expresses the correspondence between parameters and expressions. This correspondence is expressed in the metamodel by each event having an ordered sequence of parameters and each push edge having an ordered sequence of expressions, with the assumption that these sequences are in one-to-one correspondence.

Note that we have chosen a single Alloy model to express both the abstract syntax and static semantics of the example language since both structural properties and well-formedness rules are expressed by constraints in the Alloy model and there is no natural separation between the two. We also note that the Alloy Analyzer allows the graphical presentation of the meta-model specified by an Alloy model such as the one below.

```
sig EPSystem {models: some EPModel} {one (models & Main)}
sig EPModel {properties: set Property, events: set Event}
  {one s:EPSystem | this in s.models}
sig Main extends EPModel { }

sig Property { type: EPModel } {one m:EPModel | this in m.properties}
pred Property::sameModel[p:Property] {this.~properties=p.~properties}

sig Event {feedsOn: set Property, params: set Parameter,
  edges: set PushEdge, impactEdges: set ImpactEdge}
fact { all e:Event | { e.feedsOn in (e.~events).properties
  all g:e.edges|{e.sameModel[g.link] && g.link.type=g.into.~events
  conformMapping[g.mappings, g.into.params]
  all m:g.mappings | { m.expr.usesProps in e.feedsOn &&
    m.expr.usesParams in e.params}
  all c: e.impactEdges | { e.sameModel[c.impact] &&
   c.expr.usesProps in e.feedsOn &&
   c.expr.usesParams in e.params }
      one m:EPModel | e in m.events }}}
pred Event::sameModel[p:Property] {this.~events = p.~properties}
pred Event::sameModel[e:Event] {this.~events = e.~events}

sig PushEdge {into:Event,link: Property,mappings:set ParameterMapping}
  { one e:Event | this in e.edges }
fun PushEdge::paramExpr[p:Parameter]: Expression {
  (p.(~param) & this.mappings).expr}
sig ImpactEdge {impact: Property,expr: Expression}
  {expr.type =impact.type && one e:Event | this in e.impactEdges}

sig Parameter{type: EPModel} { one e:Event | this in e.params }
sig ParameterMapping {param: Parameter,expr: Expression}
  { expr.type=param.type }
pred conformMapping(m:set ParameterMapping, x: set Parameter) {
    # m = # x && m.param = x }

sig Expression { type: EPModel, usesProps: set Property,
  usesParams: set Parameter }
```

4.3 Dynamic Semantics

To describe the dynamic semantics, we first introduce in the Alloy model below the notion of state (*State signature*) and instance (*EPInstance* signature): we note that an instance has a type (an *EPModel*) and assigns for each state to each property at most one value. The *neighbors* field of *EPInstance* comprises all instances referred to by a property of this instance in a given state. The value of a property is either null or another instance; this is expressed in the Alloy model by *Value* being an abstract signature (i.e., no element can have this type) and *EPInstance* and *NullValue* being the only concrete subsignatures (i.e., subsets) of *Value*. We abstract from the expression language using the *ExpressionValue* signature, which gives the value (*val* field) of an expression (*expr* field), given the values for the properties and parameters used by the expression (*settings* field).

The actual behavior of the system is specified in the *step* predicate. This predicate expresses the fact that state $s2$ results from state $s1$ by triggering *instance event i*. The instance event (signature *InstanceEvent*) contains the information which event is triggered on which instance and what the parameter values are for a given state. To compute the effect of the instance event, it suffices to consider all instance events (including this one) that are direct or indirect successors of this instance event via push edges. This is expressed by computing the reflexive and transitive closure of the successor relation *succ* of signature *InstanceEvent* for state $s1$. This closure is denoted by the *scope* variable in the *step* predicate (line 54). We then consider all instances having an associated instance event in the scope and restrict our attention to those instance events in the scope associated with each such instance: we denote this subset of the scope by the variable *iScope* (line 55). To express the new state $s2$, we evaluate for each property that is the target of an impact edge originating from an instance event in *iScope* the expression associated with this edge (using the *exprVal* function) and state that the value of this property in state $s2$ is equal to this value (lines 58-59). All properties not targeted by impact edges in *iScope* (lines 56-57) as well as properties of instances not in the scope (lines 60-61) have the same value in $s2$ and $s1$.

```
1   open  meta_small
2   open util/ordering[State] as SO -- ordered set of states
3   sig State { }
4   pred State::init { MainInstance::defaultVal[this] }
5   sig EPInstance extends Value {
6     type: EPModel,
7     neighbors: EPInstance -> State,
  valuations: State -> Property -> Value }
{ all s: State | {
  this.reachable[s] => all p: type.properties|one this.val[p,s]
  neighbors.s = {x: EPInstance | some p: type.properties |
    s.valuations[p] = x } }}
fact {  all s: State| all y: EPInstance | all ev:Event |
    ( y.reachable[s] && ev in y.type.events ) =>
      some i: InstanceEvent | ( i.e =ev && i.x =y ) }
fun EPInstance::val [p:Property,s:State]:Value{p.(this.valuations[s])}
pred EPInstance::defaultVal[s: State] {
  all p: this.type.properties | this.val[p,s] = NullValue }
pred EPInstance::reachable[s: State]{
  this in MainInstance.*(neighbors.s) }
one sig MainInstance extends EPInstance { }{type = Main}

abstract sig Value{}
sig NullValue extends Value { }
sig ExpressionValue  {
  expr: Expression,
  setting: (Property + Parameter) -> lone Value,
  val: Value  }
{ setting.Value & Property = expr.usesProps
  setting.Value & Parameter = expr.usesParams }

sig InstanceEvent { e: Event, x: EPInstance,
  v: State-> Parameter -> lone Value,
  succ: InstanceEvent -> State }
{ let exprs = e.impactEdges.expr + e.edges.mappings.expr |
    all s: State | all ex:exprs | one exprVal[this,s,ex]
  e.~events = x.type
  all s: State | {s.v.Value = e.params &&
    {all j: succ.s|some g: e.edges| j.successorOf[this,s,g]} &&
    { all g: e.edges  | let v = x.val[g.link,s]|
      v != NullValue => some j: succ.s| j.successorOf[this,s,g]}}}
pred InstanceEvent::paramsOk[i:InstanceEvent, g: PushEdge, s:State] {
  all p:i.e.params | let ex = g.paramExpr[p] |
    p.(s.(i.v)) = exprVal[this, s, ex].val }
pred InstanceEvent::successorOf(i:InstanceEvent,s:State,g:PushEdge){
  g in i.e.edges && this.paramsOk[i,g,s] &&
      this.e = g.into && this.x = (i.x).val[g.link,s] }
fun InstanceEvent::exprVal[s: State,e: Expression]:ExpressionValue{
  { ev:ExpressionValue| ev.expr = e  &&
    (all p: e.usesParams  | ev.setting[p] = p.(s.(this.v))) &&
    (all p: e.usesProps  | ev.setting[p] = this.x.val[p,s]) }}

pred step[s1: State, i: InstanceEvent,  s2: State] {
  let scope = i.*(succ.s1) | {
    all y: scope.x | let iScope = (y.~x) & scope | {
      all p: y.type.properties| { p not in iScope.e.impactEdges.impact
        => y.val[p,s2] = y.val[p,s1]}
      all j: iScope | all g: j.e.impactEdges |
          y.val[g.impact,s2] = exprVal[j,s1,g.expr].val }
    all y: EPInstance-scope.x | all p: y.type.properties  |
      y.val[p,s2] = y.val[p,s1] }}
```

5 Discussion

We now compare traditional approaches presented in section 3 to the Alloy-based approach described in the last section. Comparing approaches implies selecting a set of criteria to base the comparison on. For this we need to first clarify the goal of the comparison. As mentioned already in the introduction many modeling languages are used without having a formal semantics. The main purpose of writing this paper is to remedy this state of affairs. Therefore it makes sense to look at those factors that have the greatest influence on the adoption of formal techniques for defining the semantics of a modeling language.

We focus on two properties that have an influence on the adoption of a formal notation (these have also been identified as key factors in the work of Mosses (eg.,[9])): (1) complexity of the notation and (2) analyzability of semantic specifications. Let us start with the notational complexity. For the abstract syntax and static semantics we saw two traditional approaches: EBNF and type checking on one side, and metamodeling on the other side. While both EBNF and metamodeling provide a succinct description of the abstract syntax, the static semantics description provided by the OCL constraints seems more accessible than the type checking approach: the more mathematical flavor of the latter notation and its higher density are probably an obstacle for a wide-spread adoption (see also [3]) in the modeling community, where this technique is less known. In our eyes both metamodeling and the Alloy approach have a similar notational complexity. In fact their close correspondence clearly comes out in the explanation of the Alloy approach in subsection 4.2 in terms of the metamodel.

For the dynamic semantics we presented only two options: operational semantics and the Alloy-based approach. We believe that that the difference between these two approaches is similar to the difference between type checking and Alloy for static semantics: the operational semantics has also a strong mathematical flavor with a very compact notation relying on many special symbols, while an Alloy model looks more like a module written in an object-oriented programming language. This should ease the adoption of Alloy in the modeling community.

If we now look at abstract syntax, static semantics and dynamic semantics as a whole, traditional approaches require at least two rather different notations, e.g., metamodeling and operational semantics, to specify the language while Alloy handles all three parts using a single notation. This is again points in favor of Alloy.

By analyzability, the second factor we want to consider, we mean the possibility to analyze the correctness of the specification using an automatic tool. Verifying the semantics of all but the most simple languages is a non-trivial task. If no tool support is provided for checking a formal description, our confidence in the correctness of the formal description is often not very strong. Immediate analyzability is also important if we want to support opportunistic design of semantics specifications, which seems to be the preferred way for humans to design complex objects (such as a formal semantics) [3].

For the traditional approaches some tool support is available: for instance we can check the metamodel with constraints using the USE tool [2]. Checking the operational semantics is usually done by proving manually or via proof assistants that some properties hold, or by implementing its rules in code and then testing or formally verifying the code. In addition to the drawback that both manual proof and code implementation can be error-prone, the diversity of the traditional approaches makes the effort of checking them automatically definitely higher than with Alloy since the latter one comes out of the box with a powerful tool, the Alloy analyzer.

Using this tool we can immediately check the correctness of the current (partial) specification (within the limits of the finite scope hypothesis). In our own experience this allowed us to find errors early in the writing of the formal semantics. Here are concrete examples of how the Analyzer helped us to write the formal specifications: first, the tool performed standard syntactic checks to reveal, for instance, incorrect use of signatures (e.g., accessing a field that does not exist in this signature); second, we checked the conformance of the static semantics in Alloy to the meta-model by formulating the OCL constraints as assertions in Alloy and looking for counterexamples using the Analyzer; third, we checked the dynamic semantics by "running" an example EP system (i.e., letting the Analyzer generate an instance satisfying the step predicate) and observing whether it "behaved" as expected.

6 Conclusion

Defining the formal semantics of a modeling language is important for reasoning about the language and for providing tool support (other arguments are given in [14]). Current approaches to formalizing semantics are often difficult to use in practice which may explain the introduction of many modeling languages that lack a formal semantics description.

In this paper we have presented a novel approach for the definition of the abstract syntax, the static semantics and the dynamic semantics of a modeling language based on the Alloy language. Two key advantages of the Alloy-based approach, which should be relevant for the applicability of the approach, are the low complexity of the notation (partly due to the fact that we need to deal with a single notation rather than several notation for the different aspects) and automatic analyzability.

Alloy was developed as a lightweight approach for developing software abstractions. It encourages incremental development of software models and reaps some of the benefits of traditional formal methods at a lower cost. Based on our experience documented in this paper we believe that the same features of Alloy also apply in the area of formal model semantics, thus opening the prospect of formal semantics becoming more accessible and more widely adopted in the definition of modeling languages.

The use of Alloy for specifying the formal semantics was illustrated in this paper on one concrete example. Future work should analyze for what type of modeling languages the Alloy based approach works well. Competing formal approaches such as those based on abstract state machines or conditional rewriting logic should also be compared with Alloy to examine the strength and weaknesses of each approach for different classes of modeling languages.

References

1. M. Broy, M. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic. 2nd UML 2 semantics symposium: Formal semantics for UML. In *Models in Software Engineering: Workshops and Symposia at MoDELS 2006*, pages 318–323. 2007.
2. M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.
3. T. R. G. Green. Cognitive dimensions of notations. In *the Proceedings of 5th conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, pages 443–460, 1989.
4. D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.
5. D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 29(4):16–30, 1996.
6. P. Kelsen. A declarative executable model for object-based systems based on functional decomposition. In *the Proceedings of the 1st International Conference on Software and Data Technologies*, pages 63–71, 2006.
7. P. Kelsen and Q. Ma. A formal definition of the EP language. Technical Report TR-LASSY-08-03, Laboratory for Advanced Software Systems, University of Luxembourg, May 2008.
8. P. Kelsen, E. Pulvermueller, and C. Glodt. Specifying executable platform-independent models using OCL. In *ECEASST*, 2008(9).
9. P. D. Mosses. Theory and practice of action semantics. In *the Proceedings of 21st Int. Symp. on Mathematical Foundations of Computer Science*, pages 37–61, 1996.
10. P. D. Mosses. The varieties of programming language semantics. In *the Proceedings of 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 165–190, 2001.
11. OMG. Unified modeling language superstructure specification 2.0, November 2005.
12. OMG. Object Constraint Language 2.0, May 2006.
13. G. Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, 1993.
14. Y. Zhang and B. Xu. A survey of semantic description frameworks for programming languages. *SIGPLAN Not.*, 39(3):14–30, 2004.