A Formal Definition of the EP Language

Pierre Kelsen and Qin Ma
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue R. Coudenhove-Kalergi, Luxembourg

TR-LASSY-08-03

May 2008

Abstract

The purpose of this paper is two-fold: first, to give a formal semantics to a declarative executable modeling language called EP; second, to introduce a new approach for defining the formal semantics of a modeling language and to compare it, in the case of the EP language, with traditional approaches for defining the formal semantics of a modeling language

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. The availability of a formal semantics is important for reasoning about the language but also for building tools for the language. In this paper we propose the Alloy language as a unified notation for this task. For the concrete example of the EP language, we contrast the Alloy 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.

Chapter 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 [14]. 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 definition of the formal semantics of a concrete language - the EP language - a declarative executable modeling language [6, 7, 8]. The term "formal semantics" usually refers to the dynamic semantics only (see, e.g., [16]). 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 of the EP language.

We first present two traditional approaches for defining abstract syntax and static semantics of EP based on EBNF and type systems on one hand, and meta-modeling on the other hand. For the 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 coined because Alloy attempts to obtain the benefits of traditional formal methods at lower cost [5].

We now discuss related work. We only present one traditional approach for specifying the dynamic semantics: operational semantics. For a fuller account of this approach the reader is referred to [16]. Other approaches for specifying the dynamic semantics are [10, 17]: 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 [17]. 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 chapter we present informally the EP language that will be used as a running example for the remainder of the paper. In Chapter 3 we describe the meta-modeling based approach for specifying the abstract syntax and static semantics of EP. In Chapter 4 we

define the abstract syntax using an EBNF grammar, the static semantics using type systems, and finally the dynamic semantics of EP by giving the operational semantics. In Chapter 5 we do the same exercise -i.e., specifying the abstract syntax, static semantics and dynamic semantics - using Alloy. We then compare in Chapter 6 the Alloy-based approach with the traditional approaches. We close with concluding remarks in the final chapter.

Chapter 2

Informal Description of the EP Language

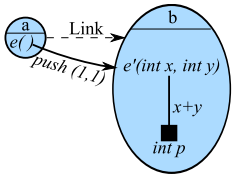
In this chapter we give an informal description of the syntax and semantics of the EP language. For a more complete description of EP the reader is referred to [6, 7, 8].

The language is summarized as a meta-model in Figure 3.1. An EP *system* defines a set of *models*, each of which consists of a set of *properties* and a set of *events* to cover both structural and behavioral characteristics. In any valid system, there should be exactly one model appointed as the main model. This main model is 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. In addition, EP *interfaces* can also be defined, to specify only the type information of properties and/or events, but without implementation details. Interfaces can extend other interfaces, and models can implement interfaces. This will build up a hierarchy of inheritance. Nominal subtyping is followed. Namely, if A inherits from B, by either extending or implementing, then for any places where a B instance is wanted, any A instance will also work.

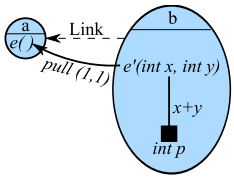
There are two types of properties in the system: *local properties* express the state of the system; *query properties* express side-effect free functions that are evaluated over the state of the system. The state of the system can be modified using events. An event can modify a local property via an *impact edge* carrying an expression that computes the new value of the impacted property. Expressions in EP models are formulated using the OCL expression language [12, 8].

Since an event may affect multiple properties in multiple models we can define event edges that will represent which other events will be executed when the current event is triggered. There are two types of *event edges*: *push edges* and *pull edges*. An event may have a number of such edges associated with itself. Both types of event edges have a target event and a link property, whose value indicates on which instance the target event will be triggered. Pull edges and push edges differ fundamentally in the way in which the flow of control is transfered: a pull edge means that the event that owns this edge will be triggered when the target event is triggered. On the other hand a push edge works in the opposite direction: when the event owning this edge is executed it triggers the target event. So for pull edges flow of execution is from the target event to this event (owning the event edge) while for push edges flow of execution is from this event to the target event. Events may have parameters; expressions on the event edges provide the values for the parameters when the system executes.

To really understand the dynamic semantics of events, it helps to look at a concrete example with model instances that exist at run-time.



We consider the situation with two instances "a" and "b". Instance "a" (i.e., the corresponding model) has an event "e" with no parameters, and instance "b" has an event "e'" that takes two integer parameters. The solid arrow denotes a push edge between events and the dashed arrow labeled "Link" specifies the link of the edge which is basically a local property of "a" that points to "b". The square-headed arrow between event "e'" and local property "p" on instance "b" denotes an impact edge, whose associated expression is $x + y$, i.e. the sum of the two parameters of the event. In a nutshell, this example depicts the following event propagations and local property modification: execution of "e" on "a" would trigger execution of "e'" on "b" with the constant arguments 1 and 1, which would in turn change the value of "p" on instance "b" to the sum of the two arguments of "e'", i.e. 2.



The same scenario can also be captured by replacing the push edge with a pull edge. However, this time, the link between the two instances becomes a local property of instance "b" that points to "a". We see that in both situations, the execution at runtime will start from e on "a" and flow to e' on "b". However, with the push edge, it is instance "a" that initiates the propagation as in "a" pushing the execution into "b"; while with the pull edge, it is instance "b" that initiates the propagation as in "b" pulling the execution from "a".

Chapter 3

Approach 1: Meta-modeling

The common practice in formally specifying a modeling language is OCL [12] constraints based meta-modeling. The meta-model we present here to define the EP language is a MOF 2.0 [11], or equivalently UML 2.0 Infrastructure [13], model.

Following the terminology in the (meta-)modeling community, the abstract syntax of the meta-model, i.e. the EP language here, consists of:

- a number of meta-classes (instances of MOF Class) together with a number of properties for each (instance of MOF Property),
- a number of bi-directional associations between meta-classes (instances of MOF Association),
- a number of generalization relations (instances of MOF Generalization),
- and finally, a number of additional well-formedness rules that are not covered by the previous items (instances of MOF Constraint).

We present the first three items above in terms of the UML class diagram in Figure 3.1, while adopting the concrete syntax of OCL as in [12] for recording additional well-formedness rules other than multiplicity or navigability constraints.

The chapter is organized as follows: for each meta-class in the class diagram, a dedicated description is specified, following the meta-class specification format given below.

UML class diagram format Names of abstract meta-model elements are written in italics. Properties of a meta-class A are shown in two ways, depending on the type. All primitive (including enumeration) typed properties are directly included in the class box for A, while for a property whose type is meta-class B, it is depicted as a uni-directional association from A to B, with the name of the property being the unique navigable association end. Operations of meta-classes are also shown inlined in the meta-class boxes.

If a meta-association end is navigable but there is no explicit role name specified, then it has an implicit role name which is the name of the meta-class being associated to the end with the first letter in lowercase. Moreover, multiplicity "*" is an equivalent short-cut for "0..*".

Meta-class specification format The technical description of a meta-class is organized into one table, which is broken down into several compartments corresponding to different aspects. In cases where a certain aspect is not applicable to a meta-class, the corresponding compartment is omitted entirely. The following compartments and conventions are used in the table to define a meta-class.

- Convention:
  - MetaClass names are presented in sans-serif font, and *AbstractMetaClass* names are italicized.

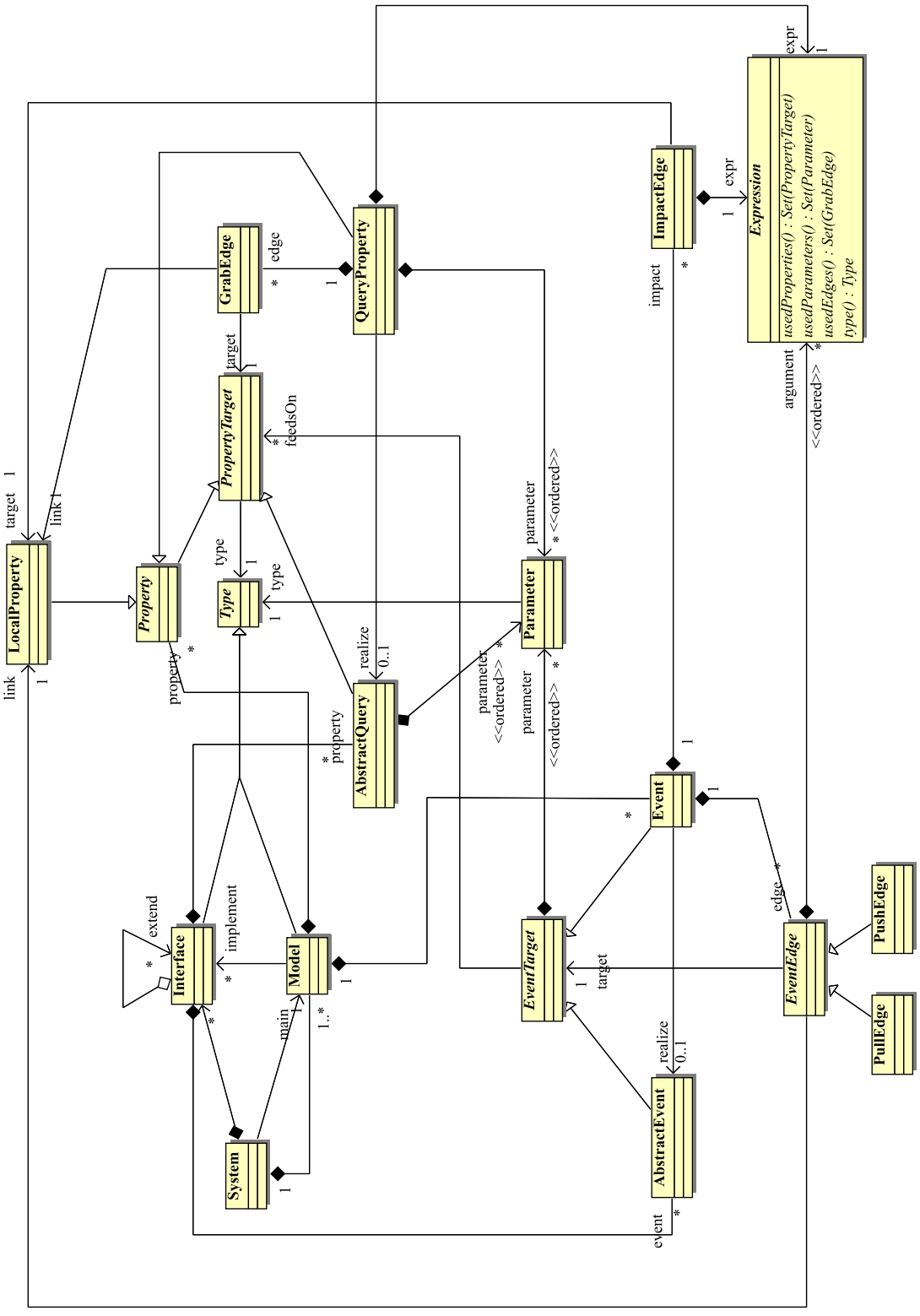


Figure 3.1: EP meta-model abstract syntax

  - Primitive typed properties (i.e. those appearing in class boxes in diagrams), are collected into the **Attribute** compartment. Meta-class typed properties (i.e. those uni-directional associations in diagrams), are collected into the **Association** compartment.
  - Bi-directional associations are described as two uni-directional associations, with each of the two association ends appearing in the **Association** compartment of the opposite class.

- The table heading gives the name of the meta-class and indicates whether it is abstract.
- The **Description** compartment gives an informal description of the meta-class, to talk about its purpose, nature, meaning, potential usage, etc.
- The **Generalizations** compartment lists all direct super-classes of the meta-class.
- The **Attributes** compartment lists each of the attributes of the meta-class in the following format:
  - ⟨Name⟩: ⟨Meta-Type⟩ [⟨Multiplicity⟩]
    ⟨Textual description of the purpose and meaning of the attribute⟩
- The **Associations** compartment lists each of the associations of the meta-class in the following format:
  - ⟨Name⟩: ⟨Meta-Type⟩ [⟨Multiplicity⟩]
    ⟨Textual description of the purpose and meaning of the association⟩

  When the association is an aggregation, we append a tailing ◊ to indicate the case as:

  - ⟨Name⟩: ⟨Meta-Type⟩ [⟨Multiplicity⟩] ◊
    ⟨Textual description of the purpose and meaning of the association⟩

  Similarly, when the association is a composition, we append a tailing ⧫ indicate the case as:

  - ⟨Name⟩: ⟨Meta-Type⟩ [⟨Multiplicity⟩] ⧫
    ⟨Textual description of the purpose and meaning of the association⟩

Constraints are then numerically listed below the table, to define the additional well-formedness rules applied to the meta-class. Each constraint consists of a textual description and a formal expression in OCL. Moreover, if some additional auxiliary operations are needed in order to define some constraints (or recursively some other operations), an additional paragraph titled "Additional operations" will appear before the "Constraints" paragraph, with the following format:

- ⟨Name⟩(⟨Parameter list⟩): ⟨Return-Meta-Type⟩
  ⟨Textual description of the purpose and meaning of the operation⟩
  ⟨Operation definition in OCL⟩

where ⟨Parameter list⟩ is either empty or pairs of parameter name and type, separated by comma.

3.1 Meta-class descriptions

System	
Associations	• model : Model[1..*] ⧫ An EP system owns a non-empty set of EP models. • interface : Interface[*] ⧫ An EP system owns a set of EP interfaces. • main : Model[1] An EP system denotes a unique main model.

Constraints System is subject to the following constraints:

1. The appointed main model of an EP system must be owned by the system.

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

Type {abstract}
Description : *Type* is an abstract meta-class. It is specialized into two meta-classes: Model and Interface.

Additional operations The following operations are defined in order to support the definition of the static semantics in OCL:

- *conformsTo(t: Type): Boolean*
  Returns true if the self type conforms to the argument type, otherwise false. To be refined in sub-classes.

Interface	
Generalizations : *Type*	
Associations	• property : AbstractQuery[*] ♦ The set of abstract queries owned by an EP interface. • event : AbstractEvent[*] ♦ The set of abstract events owned by an EP interface. • extend : Interface[*] ◊ The set of interfaces extended by the EP interface.

Additional operations The following operations are defined in order to support the definition of the static semantics in OCL:

- superInterfaces(): Set(Interface)
  Returns the set of interfaces that are extended by the interface, directly, or recursively.

```
context Interface::superInterfaces(): Set(Interface)
  body: extend->iterate(
          i: Interface;
          result:Set(Interface)=Set{} |
          result->union(i.superInterfaces())
        )->union(extend)
```

- conformsTo(t: Type): Boolean
  The interface conforms to argument type t, if and only if t is a super interface.

```
context Interface::conformsTo(t: Type):Boolean
  body: self.superInterfaces()->includes(t) or
        self = t
```

PropertyTarget {abstract}	
Associations	• type : *Type*[1] A property target has a type.

AbstractQuery

AbstractQuery *continued*	
Generalizations : *PropertyTarget*	
Associations	• interface : Interface[1] The interface that owns the abstract query. • parameter : Parameter[*] ♦ { ordered } The set of parameters of the abstract query.

EventTarget {abstract}	
Associations	• feedsOn : *PropertyTarget*[*] The set of properties that are visible in the event. • parameter : Parameter[*] ♦ { ordered } An event target owns a set of parameters.

AbstractEvent	
Generalizations : *EventTarget*	
Associations	• interface : Interface[1] The interface that owns the abstract event.

Constraints AbstractEvent is subject to the following constraints:

1. The set of feedsOn abstract queries should all be owned by the interface that owns the abstract event.

```
context AbstractEvent inv validFeedsOn:
  interface.property->includesAll(feedsOn)
```

Model	
Generalizations : *Type*	
Associations	• system : System[1] The EP system that owns the EP model. • property : *Property*[*] ♦ The set of properties, including both local properties and query properties, owned by an EP model. • event : Event[*] ♦ The set of events owned by an EP model. • implement : Interface[*] The set of interfaces implemented by the EP model.

Additional operations The following operations are defined in order to support the definition of the static semantics in OCL:

- superInterfaces(): Set(Interface)
  Returns the set of interfaces that are implemented by the model, directly, or recursively.

```
context Model::superInterfaces(): Set(Interface)
  body: implement->iterate(
          i: Interface;
          result:Set(Interface)=Set{} |
          result->union(i.superInterfaces())
        )->union(implement)
```

- conformsTo(t: Type): Boolean
  The model conforms to argument type t, if and only if t is a super interface.

```
context Model::conformsTo(t: Type):Boolean
  body: self.superInterfaces()->includes(t) or
        self = t
```

Constraints Model is subject to the following constraints:

1. For every abstract query (or abstract event) that is declared in one of the super interfaces, there exist a unique query (or event) owned by the model, which realize it.

```
context Model inv realizingAll:
  superInterfaces()->forAll(i |
    i.property->forAll(qdl |
      property->unique(qdf |
        qdf.oclIsTypeOf(QueryProperty) and
        qdf.realize = qdl
      )
    ) and
    i.event->forAll(edl |
      event->unique(edf |
        edf.realize = edl
      )
    )
  )
```

Property {abstract}	
Generalizations : *PropertyTarget*	
Associations	• model : Model[1] The EP model that owns the property.

LocalProperty
Generalizations : *Property*

QueryProperty
Generalizations : *Property*

QueryProperty *continued*	
Associations	• edge : GrabEdge[*] ♦ The set of grab edges of the query. • expr : *Expression*[1] ♦ The body expression of the query. • parameter : Parameter[*] ♦ { ordered } The set of parameters of the query. • realize : AbstractQuery[0..1] A query may realize an abstract query declared in an interface.

Constraints QueryProperty is subject to the following constraints:

1. The set of used parameters of the associated expression is included in the set of parameters of the query.

```
context QueryProperty inv usedParameter:
  parameter->includesAll(expr.usedParameters())
```

2. The set of used properties of the associated expression is included in the set of properties owned by the model that owns the query.

```
context QueryProperty inv usedProperty:
  model.property->(expr.usedProperties())
```

3. The set of used edges of the associated expression should be included in the set of grab edges owned by the property.

```
context QueryProperty inv usedEdge:
  edge->includesAll(expr.usedEdges())
```

4. The type of the associated expression conforms to the declared type of the property.

```
context QueryProperty inv validType:
  expr.type().conformsTo(type)
```

5. The realized abstract query must be owned by a super interface of the model that owns the query. Moreover, the type of the query should conform to the type of the realized abstract query.

```
context QueryProperty inv validRealize:
  not realize.oclIsUndefined() implies
    model.superInterfaces()->includes(realize.interface) and
    type.conformsTo(realize.type) and
    parameter->size() = realize.parameter->size() and
    realize.parameter->forAll(para |
      para.type.conformsTo(
        (parameter->at(realize.parameter->indexOf(para))).type
      )
    )
```

Parameter	
Associations	• type : *Type*[1] The type of the parameter.

GrabEdge	
Associations	• target : *PropertyTarget*[1] The target that is being grabbed. • link : LocalProperty[1] ♦ The link property carries an instance following which we reach the target. • queryProperty : QueryProperty[1] The query that owns the grab edge.

Constraints GrabEdge is subject to the following constraints:

1. The link property must be owned by the same model that owns the query property that owns the edge.

```
context GrabEdge inv validLink:
  property.model.property->includes(link)
```

2. The target must be owned by the type of the link property.

```
context GrabEdge inv validTarget:
  link.type.oclIsTypeOf(Interface) implies
    link.type.oclAsType(Interface).property
      ->includes(target) and
  link.type.oclIsTypeOf(Model) implies
    link.type.oclAsType(Model).property
      ->includes(target)
```

Event	
Generalizations : *EventTarget*	
Associations	• model : Model[1] The EP model that owns the event. • edge : *EventEdge*[*] ♦ The set of event edges (push or pull edges) of the event. • impact : ImpactEdge[*] ♦ The set of impact edges of the event. • realize : AbstractEvent[0..1] An event may realize an abstract event declared in an interface.

Constraints Event is subject to the following constraints:

1. The set of feedsOn properties should all be owned by the model that owns the event.

```
context Event inv validFeedsOn:
  model.property->includesAll(feedsOn)
```

2. The realized abstract event must be owned by a super interface of the model that owns the event. Moreover, the type of the event should conform to the type of the realized abstract event.

```
context Event inv validRealize:
  not realize.oclIsUndefined() implies
```

```
model.superInterfaces()->includes(realize.interface) and
parameter->size() = realize.parameter->size() and
realize.parameter->forAll(para |
  para.type.conformsTo(
    (parameter->at(realize.parameter->indexOf(para))).type
  )
)
```

EventEdge {abstract}	
Description : *EventEdge* is an abstract meta-class. It is specialized into two meta-classes: PushEdge and PullEdge.	
Associations	• event : Event[1] The event that owns the edge. • target : EventTarget[1] The target that the edge pushes execution into or pull from. • argument : *Expression*[*] ♦ { ordered } The ordered set of arguments pushed or pulled by the edge. • link : LocalProperty[1] ♦ The link property carries an instance following which we reach the target.

Constraints EventEdge is subject to the following constraints:

1. The link property must be owned by the model that owns the event that owns the edge.

```
context EventEdge inv validLink:
  event.model.property->includes(link)
```

2. The target must be owned by the type of the link property.

```
context EventEdge inv validTarget:
  link.type.oclIsTypeOf(Interface) implies
    link.type.oclAsType(Interface).event
      ->includes(target) and
  link.type.oclIsTypeOf(Model) implies
    link.type.oclAsType(Model).event
      ->includes(target)
```

3. The sets of used edges of the argument expressions of the edge are all empty.

```
context EventEdge inv usedEdge:
  argument->forAll(a |
    a.usedEdges()->isEmpty()
  )
```

PushEdge
Generalizations : *EventEdge*

Constraints PushEdge is subject to the following constraints:

1. The sets of used properties of the argument expressions of the edge should all be included in the set of feeds on properties of the event that owns the edge.

```
context PushEdge inv usedProperty:
  argument->forAll(a |
    event.feedsOn->includesAll(a.usedProperties())
  )
```

2. The sets of used parameters of the argument expressions of the edge should all be included in the set of parameters of the event that owns the edge.

```
context PushEdge inv usedParameter:
  argument->forAll(a |
    event.parameter->includesAll(a.usedParameters())
  )
```

3. The types of the argument expressions should respectively conforms to the types of the parameters of the target event of the edge.

```
context PushEdge inv validType:
  argument->forAll(a |
    a.type().conformsTo(
      (target.parameter->at(argument->indexOf(a))).type
    )
  )
```

PullEdge
Generalizations : *EventEdge*

Constraints PullEdge is subject to the following constraints:

1. The sets of used properties of the argument expressions of the edge should all be included in the set of feeds on properties of the target event of the edge.

```
context PullEdge inv usedProperty:
  argument->forAll(a |
    target.feedsOn->includesAll(a.usedProperties())
  )
```

2. The sets of used parameters of the argument expressions of the edge should all be included in the set of parameters of the target event of the edge.

```
context PullEdge inv usedParameter:
  argument->forAll(a |
    target.parameter->includesAll(a.usedParameters())
  )
```

3. The types of the argument expressions should respectively conform to the types of the parameters of the event that owns the edge.

```
context PushEdge inv validType:
  argument->forAll(a |
    a.type()=(event.parameter->at(argument->indexOf(a))).type
  )
```

ImpactEdge

ImpactEdge *continued*	
Associations	• target : LocalProperty[1] The local property that is impacted by the edge, i.e. whose value is changed according to the associated expression of the edge. • expr : *Expression*[1] ♦ The expression associated to the edge in order to compute the new value for the target property. • event : Event[1] The event that owns the edge.

Constraints ImpactEdge is subject to the following constraints:

1. The target property should be owned by the model that owns the event that owns the edge.

```
context ImpactEdge inv validTarget:
  event.model.property->includes(target)

endpackage
```

2. The set of used properties of the associated expression should be included in the properties of the owning model of the event that owns the edge.

```
context ImpactEdge inv usedProperty:
    event.model.property->includesAll(expr.usedProperties())
```

3. The set of used parameters of the associated expression should be included in the parameters of the event that owns the edge.

```
context ImpactEdge inv usedParameter:
    event.parameter->includesAll(expr.usedParameters())
```

4. The set of used edges of the associated expression should be empty.

```
context ImpactEdge inv usedEdge:
    expr.usedEdges()->isEmpty()
```

5. The type of the associated expression should conforms to the type of the target property.

```
context ImpactEdge inv validType:
  expr.type().conformsTo(target.type)
```

Expression {abstract}
Description : *Expression* is an abstract meta-class. It would be specialized into sub-classes in an accompanying expression language, to denote various kinds of expressions such as literal expressions, variable expressions, call expressions, instance create expressions, if-then-else expressions, and let binding expressions.

Expression *continued*	
Operations	• *usedProperties():Set(PropertyTarget)* Returns the set of properties that are referenced by the expression. This abstract operation would be redefined in sub-classes. • *usedParameters():Set(Parameter)* Returns the set of parameters that are referenced by the expression. This abstract operation would be redefined in sub-classes. • *usedEdges():Set(GrabEdge)* Returns the set of grab edges that are referenced by the expression. This abstract operation would be redefined in sub-classes. • *type():Type* Returns the type of the expression. This abstract operation is redefined in sub-classes.

Chapter 4

Approach 2: EBNF, Type Systems, and Operational Semantics

Another way to formalize EP is to follow the approach usually adopted for specifying formal languages, namely via EBNF based syntax, type checking rules, and operational semantics. We discuss this approach in this chapter.

4.1 EBNF based abstract syntax

The EBNF based abstract syntax appears in Figure 4.1. Different from the EP meta-model 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 $x, y, z \in \mathcal{X}$; property names denoted by η including local property names $p \in \mathcal{P}$ and query property names $q \in \mathcal{Q}$; event names $e \in \mathcal{E}$; interface names $f, g, h \in \mathcal{F}$; and model names $m, n \in \mathcal{M}$. Moreover, we write $\overline{\texttt{XX}}$ for a sequence of form $\texttt{XX}_1, \ldots, \texttt{XX}_k$. An alternative notation is $\texttt{XX}_i\,^{1 \leq i \leq k}$, or simply $\texttt{XX}_i\,^{i \in I}$ where I stands for the set of possible subscripts. An empty sequence is denoted by ϵ. We also abbreviate operations on sequences in a obvious way. For example, we write $\overline{m}\_\overline{x}$ as shorthand for $m_1\_x_1, \ldots, m_k\_x_k$, and $\overline{y} : \mathsf{ptype}[m,\ \overline{\eta}]$ for $y_1 : \mathsf{ptype}[m,\ \eta_1], \ldots, y_k : \mathsf{ptype}[m,\ \eta_k]$.

4.2 Typing

In this section we assume that the reader is familiar with type systems [15]. Type checking rules are defined for EP to ensure the well-formedness of EP systems, the intended scope of names, and the correct runtime behavior of EP systems in the sense that no "name undefined" and no "incompatible type" would be reported during the execution of type checked systems.

However, detecting cyclic property definitions (i.e., the initialization of a property relies on itself, or the computation of a query involves infinitely recursively calling itself), or cyclic event definitions (i.e. the occurrence of an event asks for infinite recursive triggering of itself), goes beyond the scope of the type system. Dedicated analysis techniques on the propagation graphs built following the edges in a system should be developed for this purpose. Similarly, the task to rule out an event definition whose execution involves impacting a property multiple times so as to assure the determinism of the EP runtime semantics, falls also outside the responsibility of type system but will be carried out by the dynamic semantics.

4.2.1 Basics

Informally, the type of a term abstracts all the necessary information that characterizes the behavior of the term. For example, the type of an expression tells you what kind of values it may yield, and what kind

`Sys`	::=	$\overline{\texttt{Idf}}$, $\overline{\texttt{Mdf}}$	EP systems
`Idf`	::=	**interface** f **extend** $\overline{f}$ $\{\overline{\texttt{Qdl}}, \overline{\texttt{Edl}}\}$	Interface definitions
`Qdl`	::=	$t\ q(\overline{t}\ \overline{x})$	Query declarations
`Edl`	::=	$\overline{q}$ **feed** $e(\overline{t}\ \overline{x})$	Event declarations
`Mdf`	::=	**model** m **implement** $\overline{f}$ $\{\overline{\texttt{Pdf}}, \overline{\texttt{Qdf}}, \overline{\texttt{Edf}}\}$	Model definitions
`Pdf`	::=	$t\ p$	Property definitions
`Qdf`	::=	$t\ q(\overline{t}\ \overline{x})$ $\{\overline{\texttt{Qdg}}\} = \texttt{Exp}$	Query definitions
`Edf`	::=	$\overline{\eta}$ **feed** $e(\overline{t}\ \overline{x})$ $\{\overline{\texttt{Edg}}\}$	Event definitions
`Qdg`	::=	**grab** η **via** p **as** x	Grab edges
`Edg`	::=	**push** $\overline{\texttt{Exp}}$ **into** e **via** p	Push edges
		pull $\overline{\texttt{Exp}}$ **from** e **via** p	Pull edges
		impact p **with** `Exp`	Impact edges
t	::=	$f \mid m$	Types

Figure 4.1: EBNF based EP abstract syntax

of operations one may carry out on it. The formal definition of types used in the type system for the EP language appears at the top of Figure 4.3. We use `Typ` to range over any types. In addition to interface and model types, denoted by t , we also include function types denoted by $\overline{t} \rightarrow t$ to capture types of queries, which basically take parameters of types $\overline{t}$, respectively, and return a value of type t. Moreover, we use the special type **ok** to denote only well-typedness in cases where no more informative type is necessary.

A typing environment, written Γ, is a finite sequence of bindings from variables, properties, or queries, to types. Given a typing environment, we can assign, in a particular setting, types to terms whose free names are all bound in the environment. The judgment that term τ has type `Typ` in environment Γ for setting γ is expressed by writing:

$$\Gamma \vdash^{\gamma} \tau :: \texttt{Typ}$$

A setting can be intuitively understood as some additional typing related information that is not (possible to be) included in the typing environment. For example, when type checking event definitions, we need to know in the definition of which model these events are defined. The name of the concerned model would then be part of the setting of the corresponding judgment.

Finally, judgments are used in typing rules of the form:

$$\frac{\text{premise} \quad \ldots \quad \text{premise}}{\text{conclusion}}\ \textsc{Rule-Name}$$

where the conclusion is a judgment, and the premises are some judgments or other conditions. The typing rule basically states that from the premises we may reach the conclusion.

4.2.2 Auxiliary functions used in typing rules

Auxiliary functions are used in specifying type checking rules. For reference purpose, we summarize them in this section.

- Function domain[_] is defined on any sequence of pairs. It returns the sequence of the first elements.
- Function name[_] is defined on any sequence of definitions or declarations, such as property definitions, event declarations. It returns the sequence of defined or declared names.
- Function qsigs[_] is defined on an interface name. It returns the sequence of query signatures of the interface as defined in the system, where the signature of a query is of the form: $(\overline{x} : \overline{t}) \rightarrow t$, from the parameters of the query to its return type.
- Similarly, function esigs[_] returns the sequence of event signatures of the interface as defined in the system, where the signature of an event is of the form: $(\overline{q},\ (\overline{x} : \overline{t}))$, in which the first part consists of the feeding queries of the event, and the second part consists of its parameters.
- Function realize[_, _] takes a sequence of definitions and a sequence of signatures as arguments. It returns true if and only if for any signature in the second argument, there is always a definition in the first argument that implements it, namely, with the same names and conforming types.
- Function super[_] takes an interface name or a model name as argument. It returns the set of super interfaces, namely, in case of an interface name, all directly or recursively extended interfaces, and in case of a model name, all directly or recursively implemented interfaces.
- Function ptypes[_] takes a model name or an interface name as argument. It returns the sequence of type bindings for all the properties and queries of the model or the interface as defined in the system. Note that the type of a query property is a function type from the types of parameters to the return type.
- Function ptype[_, _] takes two arguments. The first is a model name or an interface name t defined in the system, and the second is a property name or a query name η. It returns the type of η as in t, or otherwise $\perp$ if η is not in t to indicate an error during type checking.
- Function esig[_, _] takes two arguments. The first is a model name or an interface name t defined in the system, and the second is an event name e. It returns the signature of event e as in t or otherwise returns $\perp$.

We present the formal definitions of qsigs[_], esigs[_], realize[_, _], super[_], ptypes[_], ptype[_, _], and esig[_, _], in Figure 4.2.

4.2.3 Typing rules

Typing rules appear as the main part of Figure 4.3. We take as granted the existence of a type system for the accompanying expression language. Briefly, we can type check an expression whenever needed and if it is well-typed we know the type. Such a statement is expressed by using typing judgments for expressions of form $\Gamma \vdash \texttt{Exp} :: t$, but without further rules to derive it.

Typing rules in principle correspond to the OCL constraints discussed in Chapter 3. Taking the meta-class QueryProperty on page 11 as an example, five constraints should be satisfied, namely invariants:

1. The set of used parameters of the associated expression is included in the set of parameters of the query.

```
context QueryProperty inv usedParameter:
  parameter->includesAll(expr.usedParameters())
```

2. The set of used properties of the associated expression is included in the set of properties owned by the model that owns the query.

```
context QueryProperty inv usedProperty:
  model.property->(expr.usedProperties())
```

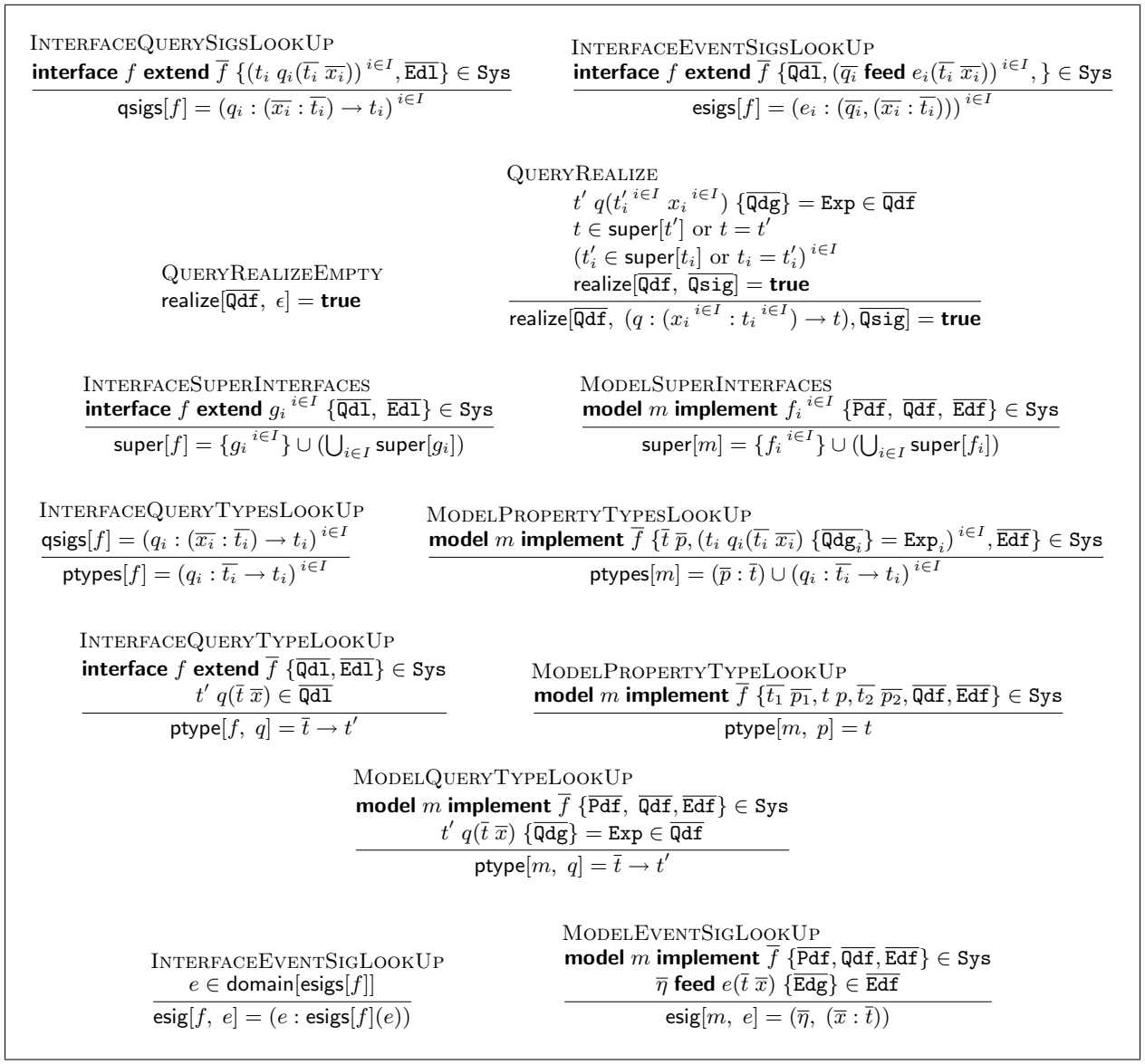


Figure 4.2: EP type system auxiliary functions

3. The set of used edges of the associated expression should be included in the set of grab edges owned by the property.

```
context QueryProperty inv usedEdge:
  edge->includesAll(expr.usedEdges())
```

4. The type of the associated expression conforms to the declared type of the property.

```
context QueryProperty inv validType:
  expr.type().conformsTo(type)
```

5. The realized abstract query must be owned by a super interface of the model that owns the query. Moreover, the type of the query should conform to the type of the realized abstract query.

Types:

$$\texttt{Typ} ::= t \mid \overline{t} \to t \mid \textbf{ok}$$

Typing environment:

$$\Gamma ::= \epsilon \mid (x : t), \Gamma \mid (x : \overline{t} \to t), \Gamma \mid (p : t), \Gamma \mid (q : \overline{t} \to t), \Gamma$$

Type checking rules:

$$\frac{\begin{array}{l}\mathsf{name}[\texttt{Mdf}_j{}^{j\in J}]\ \text{distinct} \\ \text{“Main”} \in \mathsf{name}[\texttt{Mdf}_j{}^{j\in J}] \qquad \mathsf{name}[\texttt{Idf}_i{}^{i=1,\ldots,r}]\ \text{distinct} \\ (\vdash^{\mathsf{name}[\texttt{Idf}_i{}^{i=1,\ldots,r}]}\ \texttt{Mdf}_j\ ::\ \textbf{ok})^{\,j\in J} \qquad \vdash^{\mathsf{name}[\texttt{Idf}_k{}^{k=1,\ldots,i-1}]}\ \texttt{Idf}_i\ ::\ \textbf{ok}\end{array}}{\vdash\ \texttt{Idf}_i{}^{i=1,\ldots,r},\ \texttt{Mdf}_j{}^{j\in J}\ ::\ \textbf{ok}}\ \textsc{TSystem}$$

$$\frac{\begin{array}{l}\mathsf{name}[\overline{\texttt{Qdl}}], \mathsf{name}[\texttt{Edl}_i{}^{i\in I}]\ \text{distinct} \\ \bigcap_{h\in\mathsf{super}[f]} \mathsf{domain}[\mathsf{qsigs}[h]] \cap \mathsf{name}[\overline{\texttt{Qdl}}] = \emptyset \qquad \overline{f} \subseteq \overline{g} \\ \bigcap_{h\in\mathsf{super}[f]} \mathsf{domain}[\mathsf{esigs}[h]] \cap \mathsf{name}[\texttt{Edl}_i{}^{i\in I}] = \emptyset \qquad (\vdash^{f}\ \texttt{Edl}_i\ ::\ \textbf{ok})^{\,i\in I}\end{array}}{\vdash^{\overline{g}}\ \textbf{interface}\ f\ \textbf{extend}\ \overline{f}\ \{\overline{\texttt{Qdl}},\ \texttt{Edl}_i{}^{i\in I}\}\ ::\ \textbf{ok}}\ \textsc{TInterface}$$

$$\frac{\overline{q} \subseteq \mathsf{domain}[\mathsf{qsigs}[f]] \qquad \overline{x}\ \text{distinct}}{\vdash^{f}\ \overline{q}\ \textbf{feed}\ e(\overline{t}\ \overline{x})\ ::\ \textbf{ok}}\ \textsc{TEventDeclaration}$$

$$\frac{\begin{array}{l}\mathsf{name}[\overline{\texttt{Pdf}}], \mathsf{name}[\texttt{Qdf}_i{}^{i\in I}], \mathsf{name}[\texttt{Edf}_j{}^{j\in J}]\ \text{distinct} \qquad \overline{f} \subseteq \overline{g} \\ \mathsf{realize}[\texttt{Qdf}_i{}^{i\in I},\ \biguplus_{h\in\mathsf{super}[m]} \mathsf{qsigs}[h]] \qquad (\vdash^{m}\ \texttt{Qdf}_i\ ::\ \textbf{ok})^{\,i\in I} \\ \mathsf{realize}[\texttt{Edf}_j{}^{j\in J},\ \biguplus_{h\in\mathsf{super}[m]} \mathsf{esigs}[h]] \qquad (\vdash^{m}\ \texttt{Edf}_j\ ::\ \textbf{ok})^{\,j\in J}\end{array}}{\vdash^{\overline{g}}\ \textbf{model}\ m\ \textbf{implement}\ \overline{f}\ \{\overline{\texttt{Pdf}},\ \texttt{Qdf}_i{}^{i\in I},\ \texttt{Edf}_j{}^{j\in J}\}\ ::\ \textbf{ok}}\ \textsc{TModel}$$

$$\frac{\begin{array}{l}(\vdash^{m}\ \texttt{Qdg}_i\ ::\ (y_i : \texttt{Typ}_i))^{\,i\in I} \\ \overline{x}, y_i{}^{i\in I}\ \text{distinct} \qquad t \in \mathsf{super}[t']\ \text{or}\ t = t' \\ (\textbf{self} : m) \uplus \mathsf{ptypes}[m] \uplus (\overline{x} : \overline{t}) \uplus (y_i : \texttt{Typ}_i)^{\,i\in I} \vdash \texttt{Exp}\ ::\ t'\end{array}}{\vdash^{m}\ t\ q(\overline{t}\ \overline{x})\ \{\texttt{Qdg}_i{}^{i\in I}\} = \texttt{Exp}\ ::\ \textbf{ok}}\ \textsc{TQuery}$$

$$\frac{\mathsf{ptype}[\mathsf{ptype}[m,\ p],\ \eta] = \texttt{Typ}}{\vdash^{m}\ \textbf{grab}\ \eta\ \textbf{via}\ p\ \textbf{as}\ x\ ::\ (x : \texttt{Typ})}\ \textsc{TGrabEdge}$$

$$\frac{\begin{array}{ll} & (\vdash^{(m,\overline{t})}\ \texttt{Edg}\ ::\ \textbf{ok})^{\,\texttt{Edg}\in\overline{\texttt{Edg}}}\ \textbf{pull} \\ \overline{x}\ \text{distinct} & ((\overline{x} : \overline{t}) \uplus (\Gamma_m \restriction \overline{\eta}) \vdash^{m} \texttt{Edg}\ ::\ \textbf{ok})^{\,\texttt{Edg}\in\overline{\texttt{Edg}}}\ \textbf{push} \\ \Gamma_m = \mathsf{ptypes}[m] & \\ \overline{\eta} \subseteq \mathsf{domain}[\Gamma_m] & ((\textbf{self} : m) \uplus (\overline{x} : \overline{t}) \uplus \Gamma_m \vdash^{m} \texttt{Edg}\ ::\ \textbf{ok})^{\,\texttt{Edg}\in\overline{\texttt{Edg}}}\ \textbf{impact}\end{array}}{\vdash^{m}\ \overline{\eta}\ \textbf{feed}\ e(\overline{t}\ \overline{x})\ \{\overline{\texttt{Edg}}\}\ ::\ \textbf{ok}}\ \textsc{TEvent}$$

$$\frac{\begin{array}{l}\mathsf{esig}[\mathsf{ptype}[m,\ p],\ e] = (\overline{\eta},\ (x_i{}^{i\in I} : t_i{}^{i\in I})) \\ (\Gamma \vdash \texttt{Exp}_i\ ::\ t_i')^{\,i\in I} \qquad (t_i \in \mathsf{super}[t_i']\ \text{or}\ t_i = t_i')^{\,i\in I}\end{array}}{\Gamma \vdash^{m}\ \textbf{push}\ \texttt{Exp}_i{}^{i\in I}\ \textbf{into}\ e\ \textbf{via}\ p\ ::\ \textbf{ok}}\ \textsc{TPushEdge}$$

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{Exp}\ ::\ t \\ \mathsf{ptype}[m,\ p] \in \mathsf{super}[t]\ \text{or}\ \mathsf{ptype}[m,\ p] = t\end{array}}{\Gamma \vdash^{m}\ \textbf{impact}\ p\ \textbf{with}\ \texttt{Exp}\ ::\ \textbf{ok}}\ \textsc{TImpactEdge}$$

$$\frac{\begin{array}{l}\mathsf{esig}[\mathsf{ptype}[m,\ p],\ e] = (\overline{\eta},\ (x_i{}^{i\in I} : t_i{}^{i\in I})) \\ ((\overline{x} : \overline{t}) \uplus (\mathsf{ptypes}[\mathsf{ptype}[m,\ p]] \restriction \overline{\eta}) \vdash \texttt{Exp}_i\ ::\ t_i')^{\,i\in I} \qquad (t_i \in \mathsf{super}[t_i']\ \text{or}\ t_i = t_i')^{\,i\in I}\end{array}}{\vdash^{(m,t_i{}^{i\in I})}\ \textbf{pull}\ \texttt{Exp}_i{}^{i\in I}\ \textbf{from}\ e\ \textbf{via}\ p\ ::\ \textbf{ok}}\ \textsc{TPullEdge}$$

Figure 4.3: EP type system

```
context QueryProperty inv validRealize:
  not realize.oclIsUndefined() implies
    model.superInterfaces()->includes(realize.interface) and
    type.conformsTo(realize.type) and
    parameter->size() = realize.parameter->size() and
    realize.parameter->forAll(para |
      para.type.conformsTo(
        (parameter->at(realize.parameter->indexOf(para))).type
      )
    )
```

Now let us look at the corresponding typing rule TQuery. In its last premise, the typing environment in which the body expression of the query is typed consists of three parts (in addition to the binding for **self**), combined by the disjoint union, denoted by $\uplus$, to enforce disjoint domains. The first part contains type bindings for all the local properties and queries defined in the model that defines this query; the second part contains type bindings for all the parameters of the query; and the third part contains grabbed properties and queries that are all referenced via their variable alias local to the query, accumulated in the first premise of the same rule while type checking the grab edges of the query. In other words, the only possible references from the body expression fall into these three categories, hence invariant 1, 2, and 3 are assured. Moreover, the third premise of rule TQuery also requires that the declared return type of the query either is a super type of, or is the same as, the type of the body expression, hence invariant 4 is also assured.

The assurance of invariant 5 is a bit subtle. The textual nature of this approach (in contrast to the meta-model and OCL based approach), forces us to treat things in a slightly different way. Here, when a model implements an interface, query definitions in the model automatically realize query declarations in the interface with the same name. In other words, the realization relation between query definition and declaration is drawn in model definitions implicitly by name equality. On the contrary, in the EP meta-model, the relation is drawn explicitly between QueryProperty and AbstractQuery. Referring to QueryProperty::validRealize, the first part of the invariant about ownerships becomes irrelevant. However, the second part that is about type conformance still needs to be checked. And this is carried out in the typing rule of model definitions TModel, by the first "realize[..., ...]" premise. According to the definition of the auxiliary function realize[_, _] in Figure 4.2, for a query definition to actually realize a query declaration, we basically require that they take the same parameters with conforming types, and also have conforming return types.

There are other difference caused by the textual nature, too. For instance, as it is no longer possible to refer to grab edges directly in expressions as in the meta-model approach, here we introduce an alias for the target grabbed properties to be used in expressions as local variables. Moreover, additional bookkeeping and name distinction premises are also required, such as the second premise of rule TProperty. 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".

Another difference involves constraints that enforce the ownership of a model to a member such as a property. For example, the two invariants of meta-class GrabEdge constrain the owner of the link property (invariant GrabEdge::validLink) and the owner of the target property (invariant GrabEdge::validTarget). In the corresponding typing rule TGrabEdge, the former is assured when computing ptype$[m,\ p]$, because if the link property p is not owned by the model m, $\perp$ will be returned. Similarly, the latter is assured when computing ptype[ptype$[m,\ p],\ \eta$], i.e., the grab target η must be defined in the type of the link property p.

We leave it up to the reader to explore more details of the EP type system while comparing it to the OCL constraints presented in the previous chapter.

4.3 Operational semantics

In this section we assume that the reader is familiar with operational semantics [16]. A system configuration (Λ, s) of EP runtime is a pair of an evaluation environment and a state. The evaluation environment

Values, states, and substitutions:

$$
\begin{array}{llll}
\text{States:} & s & ::= & \emptyset \mid (id : (m, \varphi_m)), s \\
\text{Values:} & \texttt{Val} & ::= & v \mid \lambda \overline{x} \rightarrow \texttt{Exp} \\
 & v & ::= & \textbf{null} \mid id \\
\text{Evaluations:} & \varphi_m & : & \mathcal{P} \rightarrow \mathcal{V} \\
\text{Substitutions:} & \sigma & : & \mathcal{ID} \times \mathcal{P} \rightarrow \mathcal{V}
\end{array}
$$

Evaluation environments:

$$\Lambda ::= \epsilon \mid (x : \texttt{Val}), \Lambda$$

External query invocations:

$$\frac{\Lambda \vDash (\texttt{Exp}.q(\overline{\texttt{Exp}}),\ s) \Downarrow (v,\ s_\Delta)}{(\Lambda,\ s) \xrightarrow{\texttt{Exp}.q(\overline{\texttt{Exp}})} (\Lambda,\ s \cup s_\Delta)} \text{QueryInvocationReduction}$$

External event invocations:

$$\frac{\Lambda \vDash (\texttt{Exp}.e(\overline{\texttt{Exp}}),\ s) \Downarrow (\sigma,\ s_\Delta)}{(\Lambda,\ s) \xrightarrow{\texttt{Exp}.e(\overline{\texttt{Exp}})} (\Lambda,\ s\sigma \cup s_\Delta)} \text{EventInvocationReduction}$$

Figure 4.4: Operational semantics: terms

binds variables to values. The state records the current set of instances.

As defined in Figure 4.4, each instance in the state s carries a distinct identity $id \in \mathcal{ID}$ bound with a pair (m, φ_m), where m is the model of the instance and φ_m gives its evaluation. We use $s(id).m$ to refer to the type/model of the instance id in state s, and $s(id).\varphi_m$ the corresponding evaluation. Values, ranged over by `Val`, are of two kinds. First, instance values, ranged over by $v \in \mathcal{V}$, are either **null** denoting the void instance for any model or id denotes an instance of a certain model. Second, function values, written $\lambda \overline{x} \rightarrow \texttt{Exp}$, where $\overline{x}$ denote the parameters of the function and `Exp` defines how to compute from the parameters to a value. Note that function values are all closed in the sense that the free names of `Exp` fall in two categories: $\overline{x}$ or **self**.

An evaluation of an instance of model m, written φ_m, is a partial function from properties to instance values, where $\forall p \in \mathcal{P}$, $\varphi_m(p) = \perp$ (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, otherwise $\perp$. Finally, system state substitution, denoted by σ, is a partial function from pairs of instance identities and properties to instance values. For a given substitution σ, $\sigma(id, p) = v$ means to change the value of p of instance id into v, and $\sigma(id, p) = \perp$ means to leave this value unchanged. As a consequence, applying the substitution to a state s, written $s\sigma$, returns a new state s' satisfying the following:

- $s'(id).m = s(id).m$;
- $s'(id).\varphi_m(p) = s(id).\varphi_m(p)$ if $\sigma(id, p) = \perp$ or otherwise $\sigma(id, p)$.

Sometimes (for instance in rule ImpactEdge of Figure 4.5), 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 $s_0(id).m = Main$ and $s_0(id).\varphi_m = \phi_{Main}$ (the default evaluation for $Main$), and a unique binding $(main : id)$ in Λ_0. Interacting with an EP system amounts to either invoking a query or an event. The reduction rule for invoking a query on an instance that is accessible from $main$ with a sequence of arguments is given in Figure 4.4. Although there might be new instance created along the invocation, denoted by s_Δ, the result system configuration in practice remains unchanged as the set of accessible instances from Λ is unchanged,

Event call evaluation:

$$\textsc{EventCall}\quad \frac{\begin{array}{l}\Lambda \vDash (\texttt{Exp},\ s) \Downarrow (id,\ s_\Delta) \qquad \Lambda \vDash (\overline{\texttt{Exp}},\ s) \Downarrow (\overline{v},\ \overline{s_\Delta}) \\ \overline{\eta}\ \textsf{feed}\ e(\overline{m}\ \overline{x})\ \{\texttt{Edg}_i^{\textsf{push}\ i\in I}, \texttt{Edg}_j^{\textsf{impact}\ j\in J}, \overline{\texttt{Edg}}^{\textsf{pull}}\} \in (s \cup s_\Delta)(id).m \\ ((\overline{x} : \overline{v}) \cup (\textsf{self} : id) \vDash (\texttt{Edg}_i^{\textsf{push}},\ s \cup s_\Delta \cup \overline{s_\Delta}) \Downarrow (\sigma_i,\ s_\Delta^i))^{\ i\in I} \\ ((\overline{x} : \overline{v}) \cup (\textsf{self} : id) \vDash (\texttt{Edg}_j^{\textsf{impact}},\ s \cup s_\Delta \cup \overline{s_\Delta}) \Downarrow (\sigma_j,\ s_\Delta^j))^{\ j\in J} \qquad \textsf{pull}[e,\ id,\ (\overline{x} : \overline{v}),\ s \cup s_\Delta \cup \overline{s_\Delta}] = (\sigma, s_\Delta^{\textsf{pull}})\end{array}}{\Lambda \vDash (\texttt{Exp}.e(\overline{\texttt{Exp}}),\ s) \Downarrow ((\biguplus_{i\in I} \sigma_i) \uplus (\biguplus_{j\in J} \sigma_j) \uplus \sigma,\ s_\Delta \cup \overline{s_\Delta} \cup (\bigcup_{i\in I} s_\Delta^i) \cup (\bigcup_{j\in J} s_\Delta^j) \cup s_\Delta^{\textsf{pull}})}$$

$$\textsc{ImpactEdge}\quad \frac{\Lambda \vDash (\texttt{Exp},\ s) \Downarrow (v,\ s_\Delta^1) \qquad \Lambda \vDash (\textsf{self},\ s) \Downarrow (id,\ s_\Delta^2)}{\Lambda \vDash (\textsf{impact}\ p\ \textsf{with}\ \texttt{Exp},\ s) \Downarrow (((id, p) : v),\ s_\Delta^1 \cup s_\Delta^2)}$$

$$\textsc{PushEdge}\quad \frac{\Lambda \vDash ((\textsf{self}.p).e(\overline{\texttt{Exp}}),\ s) \Downarrow (\sigma,\ s_\Delta)}{\Lambda \vDash (\textsf{push}\ \overline{\texttt{Exp}}\ \textsf{into}\ e\ \textsf{via}\ p,\ s) \Downarrow (\sigma,\ s_\Delta)}$$

$$\textsc{PullEffectEvaluation}\quad \frac{\begin{array}{l}\forall i \in I \text{ such that } id_i \in s, \\ \forall j \in J \text{ such that } \overline{\eta_j}\ \textsf{feed}\ e_j(\overline{m_j}\ \overline{x_j})\ \{\overline{\texttt{Edg}_j}\} \in s(id_i).m, \\ \forall k \in K \text{ such that } \textsf{pull}\ \overline{\texttt{Exp}_k}\ \textsf{from}\ e_k\ \textsf{via}\ p_k \in \overline{\texttt{Edg}_j}, \\ \text{if } e_k = e \text{ and } s(id_i).\varphi_m(p_k) = id \\ \text{then } (\overline{x} : \overline{v}) \cup (\textsf{self} : id) \vDash (id_i.e_j(\overline{\texttt{Exp}_k}),\ s) \Downarrow (\sigma_{i,j,k},\ s_\Delta^{i,j,k}); \\ \text{else } \sigma_{i,j,k} = \emptyset \text{ and } s_\Delta^{i,j,k} = \emptyset\end{array}}{\textsf{pull}[e,\ id,\ (\overline{x} : \overline{v}),\ s] = (\biguplus_{i\in I, j\in J, k\in K} \sigma_{i,j,k},\ \bigcup_{i\in I, j\in J, k\in K} s_\Delta^{i,j,k})}$$

Property call evaluation:

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

$$\textsc{QueryCallGeneral}\quad \frac{\begin{array}{l}\Lambda \vDash (\texttt{Exp},\ s) \Downarrow (id,\ s_\Delta) \\ \Lambda \vDash (\overline{\texttt{Exp}},\ s) \Downarrow (\overline{v},\ \overline{s_\Delta}) \\ n\ q(\overline{n}\ \overline{x})\ \{\overline{\texttt{Qdg}}\} = \texttt{Exp}' \in (s \cup s_\Delta)(id).m \\ (\textsf{self} : id) \vDash (\overline{\texttt{Qdg}},\ s \cup s_\Delta) \Downarrow (\overline{y} : \overline{\texttt{Val}},\ \emptyset) \\ (\textsf{self} : id) \cup (\overline{x} : \overline{v}) \cup (\overline{y} : \overline{\texttt{Val}}) \vDash (\texttt{Exp}',\ s \cup s_\Delta \cup \overline{s_\Delta}) \Downarrow (v,\ s'_\Delta)\end{array}}{\Lambda \vDash (\texttt{Exp}.q(\overline{\texttt{Exp}}),\ s) \Downarrow (v,\ s \cup s_\Delta \cup \overline{s_\Delta} \cup s'_\Delta)}$$

$$\textsc{GrabEdgeProperty}\quad \frac{\begin{array}{l}\Lambda \vDash (\textsf{self}.p_l,\ s) \Downarrow (id,\ \emptyset) \\ s(id).\varphi_m(p) = v\end{array}}{\Lambda \vDash (\textsf{grab}\ p\ \textsf{via}\ p_l\ \textsf{as}\ x,\ s) \Downarrow (x : v,\ \emptyset)}$$

$$\textsc{GrabEdgeQuery}\quad \frac{\begin{array}{l}\Lambda \vDash (\textsf{self}.p_l,\ s) \Downarrow (id,\ \emptyset) \\ n\ q(\overline{n}\ \overline{x})\ \{\overline{\texttt{Qdg}}\} = \texttt{Exp} \in s(id).m \\ (\textsf{self} : id) \vDash (\overline{\texttt{Qdg}},\ s) \Downarrow (\overline{y} : \overline{\texttt{Val}},\ \emptyset)\end{array}}{\Lambda \vDash (\textsf{grab}\ q\ \textsf{via}\ p_l\ \textsf{as}\ x,\ s) \Downarrow (x : \lambda\overline{x} \to \texttt{Exp}\{\overline{y}/\overline{\texttt{Val}}\},\ \emptyset)}$$

Expression evaluation (abridged):

$$\textsc{Variable}\quad \frac{\Lambda(x) = \texttt{Val}}{\Lambda \vDash (x,\ s) \Downarrow (\texttt{Val},\ \emptyset)}$$

$$\textsc{InstanceCreation}\quad \frac{id \text{ fresh}}{\Lambda \vDash (m :: \textsf{create}(),\ s) \Downarrow (id,\ (id : (m, \phi_m)))}$$

$$\textsc{LetBinding}\quad \frac{\Lambda \vDash (\texttt{Exp}_1,\ s) \Downarrow (v_1,\ s_\Delta^1) \qquad \Lambda \cup (x : v_1) \vDash (\texttt{Exp}_2,\ s \cup s_\Delta^1) \Downarrow (v_2,\ s_\Delta^2)}{\Lambda \vDash (\textsf{let}\ x : \texttt{Typ} = \texttt{Exp}_1\ \textsf{in}\ \texttt{Exp}_2,\ s) \Downarrow (v_2,\ s_\Delta^1 \cup s_\Delta^2)}$$

$$\textsc{PropertyCall}\quad \frac{\Lambda \vDash (\textsf{self}.p,\ s) \Downarrow (v,\ s_\Delta)}{\Lambda \vDash (p,\ s) \Downarrow (v,\ s_\Delta)}$$

$$\textsc{QueryCall}\quad \frac{\Lambda \vDash (\textsf{self}.q(\overline{\texttt{Exp}}),\ s) \Downarrow (v,\ s_\Delta)}{\Lambda \vDash (q(\overline{\texttt{Exp}}),\ s) \Downarrow (v,\ s_\Delta)}$$

$$\textsc{GrabbedNameCall}\quad \frac{\begin{array}{l}\Lambda(x) = \lambda\overline{y} \to \texttt{Exp}' \qquad \Lambda \vDash (\overline{\texttt{Exp}},\ s) \Downarrow (\overline{v},\ \overline{s_\Delta}) \\ \Lambda \cup (\overline{y} : \overline{v}) \vDash (\texttt{Exp}',\ s \cup \overline{s_\Delta}) \Downarrow (v,\ s_\Delta)\end{array}}{\Lambda \vDash (x(\overline{\texttt{Exp}}),\ s) \Downarrow (v,\ s_\Delta \cup \overline{s_\Delta})}$$

Figure 4.5: Operational semantics: rules

nor have the evaluations of instances in s changed. By contrast, invoking an event on an instance that is accessible from *main* would effectively change the system into a new configuration. The formal reduction rule is also given in Figure 4.4. Note that the evaluation environment Λ still remains the same after the reduction, because invoking an event would not change the set of global variables nor their bindings. However, the set of accessible instances from *main* may still grow, by changing the values of the properties of the main instance which are required by the substitution σ that is applied to s. However, evaluation environments may change in rules when the context in which a considered expression is evaluated changes into a local one, such as in let-binding expressions (see rule LetBinding), or for evaluating query body expressions (see rule QueryCallGeneral)in Figure 4.5.

Let us elaborate on the evaluation rules for event calls in Figure 4.5. 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 course. In rule EventCall we first evaluate the callee `Exp` into an instance value, which should not be **null** in order for the calling to be meaningful, hence is an instance identity id. Then, the arguments $\overline{\texttt{Exp}}$ are evaluated respectively. We pick up the body of the called event from the definition of $(s \cup s_\Delta)(id).m$, which effectively denotes the model of the callee, and evaluate the body, i.e., the event edges. Different kinds of event edges are evaluated separately.

Push edges and impact edges are computed in the new evaluation environment consisting of the parameters all bound to the just computed argument values, and **self** bound to the callee instance. Note that although in expressions, all reference to properties or queries come alone without any preceding dot quantification, they are all meant to be the references to **self**, as we interpret in rules PropertyCall and QueryCall. And one single binding for **self** suffices to provide the bindings for any of these properties or queries, by just following the instance evaluation in the bound value. Evaluation judgments for push or impact edges take the form: $\Lambda \vDash (\texttt{Edg},\ s) \Downarrow (\sigma,\ s_\Delta)$. In rule ImpactEdge evaluating an impact edge would result in a substitution of form $((id, p) : v)$, which basically asks to change the value of the impacted property p on instance denoted by **self** (of identity id) into the new value (i.e. v) computed from the expression `Exp`. By contrast, the result of evaluating a push edge (rule PushEdge) 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.

Computation of pull edges is a bit different since when an event e is invoked on an instance id with arguments $(\overline{x} : \overline{v})$, any instance id_i in the current state may pull the execution from id, as long as id_i has an event e_j that has a pull edge of the form **pull** $\overline{\texttt{Exp}_k}$ **from** e_k **via** p_k in which $e_k = e$ and the property p_k of id_i computes to id. As a result the child event e_j will be called on id_i with arguments $\overline{\texttt{Exp}_k}$ in the same state and the effect contributes to the total effect of calling e on id. We formalize the above behavior by an auxiliary function $\mathsf{pull}[e,\ id,\ (\overline{x} : \overline{v}),\ s]$, which is also defined in Figure 4.5 and is called in the last premise of rule EventCall.

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, with each other, nor with the original state. Therefore, the plain 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 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 effects by applying to the original state and reach a new state: $s\sigma \cup s_\Delta$.

For property or query calls, the evaluation rules appear in the middle of Figure 4.5. The general idea behind it is similar to event call evaluations, except for some details. For example, calling a query will result in a value (instead of a substitution in the case of event calls), which is basically computed by referring to the arguments, the grabbed names, and other local query or properties (see rule QueryCallGeneral). Moreover, the result of computing a grab edge whose target is a query is a function value, as shown in rule GrabEdgeQuery. We encourage the readers to explore more details.

Chapter 5

Approach 3: Alloy

5.1 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 analysis 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 analysis 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 chapter we present the Alloy models representing the syntax and semantics of the EP 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.

5.2 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 3.1. Each of the classes in the meta-model has a corresponding signature with the same name in the Alloy model. For instance, the *System* class is represented by the *System* signature. Associations in the meta-model are usually represented by fields of a signature: thus the field *models* in *System* corresponds to the aggregation from the *System* class to the *Model* class in the meta-model. Multiplicities on the association ends in the meta-model 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:System | this in s.models}` in signature *Model*. There are two signatures in the Alloy model that do not correspond to classes of the meta-model: the *Main* signature expresses the fact that there is a single distinguished model called *Main*: this is expressed in the meta-model by an association from *System* to *Model*. The signature *ParameterMapping* expresses the correspondence between parameters and expressions. This correspondence is expressed in the meta-model using ordered sequences of parameters and expressions that are constrained to be 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 EP 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.

```
1  -- this module represents the metamodel of the EP language
2  -- it describes the abstract syntax and static semantics
3  module ep/meta
4
5  -- SYSTEM, MODELS and INTERFACES
6  sig System {
7    models: some Model,
8    interfaces: set Interface
9  }{one (models & Main)} -- every EPSystem contains exactly one Main model
10
11 sig Model extends Type {
12   properties: set Property,
13   events: set Event,
14   implements: set Interface}
15 {   one s:System | this in s.models
16   all i:Interface | ( i in modelSuperTypes[this] => this.implementsInterface[i] ) }
17 pred Model::implementsInterface (i:Interface) {
18   all q:i.abstractQueries| one p: this.properties| p.realizes = q
19   all e:i.abstractEvents| one e1: this.events| e1.realizes = e }
20 fun Model::concreteEvent[e: EventTarget]: Event {
21   e in Event => e else e.~realizes & this.events }
22 fun Model::concreteQuery[q: PropertyTarget] : QueryProperty {
23   q in Property => q else q.~realizes & this.properties }
24 sig Main extends Model {}
25 sig Interface extends Type {
26   abstractQueries: set AbstractQuery,
27   abstractEvents: set AbstractEvent,
28   extend: set Interface
29 } { one s:System | this in s.interfaces }
30 fact acyclicExtend { all i: Interface|  i not in i.^extend }
31
32 -- PROPERTIES
33 abstract sig PropertyTarget { type: Type} -- property or abstractquery
34 abstract sig Property extends PropertyTarget{ } -- superclass of local and query properties
35 { one m: Model | this in m.properties }
36 fun Property::model: Model { -- returns model containing this property
37   this.~properties }
38 pred Property::sameModel[p:Property] { -- true if both properties belong to the same model
39   this.~properties = p.~properties }
40 sig LocalProperty extends Property {}
41 sig QueryProperty extends Property {
42   edges: set GrabEdge,
43   expr:  Expression,
44   params: set Parameter,
45   realizes: lone AbstractQuery }
46 { all g: edges | {
47       -- link property in same model as p
48       this.sameModel[g.link]
49       -- type of link property is equal to model containing target property
50       g.link.@type = g.targetContainer
51   }
52   expr.type.conformType[type]
53   all q:Property| q in expr.usesProps => this.sameModel[q]
54   expr.usesParams = params
55   expr.usesEdges = edges
56   some realizes => ( params = realizes.params && type = realizes.@type) }
```

```
57  sig AbstractQuery extends PropertyTarget { params: set Parameter }
58
59  -- EVENTS
60  -- eventtarget is event or abstractevent
61  abstract sig EventTarget { params: set Parameter, feedsOn: set PropertyTarget }
62  sig Event extends EventTarget {
63     edges: set EventEdge, impactEdges: set ImpactEdge, realizes: lone AbstractEvent }
64  {   feedsOn in (this.~events).properties
65      all g: edges | {
66        this.sameModel[g.link] -- link property in same model as e
67        -- type of link property is equal to model containing target event
68        g.link.type = g.targetContainer
69        conformMapping[g.mappings, g.target.@params]
70        all m:g.mappings | {
71          m.expr.usesProps in feedsOn
72          m.expr.usesParams in params
73        }
74      all c: impactEdges | {
75        this.sameModel[c.target]
76        all p: c.expr.usesProps| this.sameModel[p]
77        c.expr.usesParams in params }
78      one m:Model | this in m.events }}
79  pred Event::sameModel[p:Property] {
80    this.~events = p.~properties }
81  pred Event::sameModel[e:Event] {
82    this.~events = e.~events }
83  sig AbstractEvent extends EventTarget { }
84
85  -- EDGES BETWEEN EVENTS AND PROPERTIES
86  sig GrabEdge {
87    target: Property+AbstractQuery, link: LocalProperty }
88  {   one p:Property | this in p.edges
89    target in AbstractQuery iff link.type in Interface } -- all property edges are grab edges
90  fun GrabEdge::targetContainer: Type {
91    this.target in Property => this.target.~properties else  this.target.~abstractQueries }
92  abstract sig EventEdge {
93    target: Event + AbstractEvent,
94    mappings: set ParameterMapping,
95    link: LocalProperty
96  }{  one e:Event | this in e.edges
97    target in AbstractEvent iff link.type in Interface }
98  -- expression for this parameter in event edge
99  fun EventEdge::paramExpr[p:Parameter]: Expression {
100   (p.(~param) & this.mappings).expr }
101 fun EventEdge::targetContainer: Type {
102   this.target in Event => this.target.~events else  this.target.~abstractEvents }
103 sig PushEdge extends EventEdge { }
104 sig PullEdge extends EventEdge { }
105
106 sig ImpactEdge { target: LocalProperty, expr: Expression }
107 {   expr.type.conformType[target.type]
108   no expr.usesEdges
109 }
110
111 -- PARAMETERS
112 sig Parameter{ type: Type }
```

```
113  sig ParameterMapping { param: Parameter,expr: Expression }
114  { expr.type.conformType[param.type] }
115
116  pred conformMapping(m:set ParameterMapping, x: set Parameter) {
117      # m = # x -- for each target parameter there is one mapping
118      m.param = x }
119
120  -- EXPRESSION LANGUAGE STARTS HERE
121  sig Type {}
122  pred Type::conformType(t:Type) {
123    this in Interface => t in interfaceSuperTypes[this]
124    this in Model => (this = t || t in modelSuperTypes[this]) }
125  -- one sig BooleanType extends Type { }
126  fun interfaceSuperTypes[i:Interface]: set Interface {
127    i.*extend }
128  fun modelSuperTypes[m:Model]: set Type {
129    m.implements.*extend }
130  sig Expression {
131    type: Type,
132    usesProps: set Property,
133    usesParams: set Parameter,
134    usesEdges: set GrabEdge }
135  pred show {} run show
136
```

5.3 Dynamic Semantics

To describe the dynamic semantics, we first introduce in the Alloy model below the notion of state (*State signature*) and instance (*Instance* signature): we note that an instance has a type (an *Model*) and assigns for each state to each property at most one value. The *neighbors* field of *Instance* comprises all instances referred to by a property of this instance in a given state. The value of a property is either null, another instance or a value of another (unspecified) type ; this is expressed in the Alloy model by *Value* being a non-abstract signature and *Instance* and *NullValue* being the two 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 state, instance (on which the expression will be evaluated) and parameters used by the expression (*ps* 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. 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 or pull 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 94). 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 96). 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 99-100). All properties not targeted by impact edges in *iScope* (lines 97-98) as well as properties of instances not in the scope (lines 101-102) have the same value in $s2$ and $s1$.

```
module ep/semantics
open  ep/meta

sig State {}
-- in init state only main instance is reachable
pred State::init { MainInstance::defaultVal[this] }

sig Instance extends Value {
  type: Model,
  -- all instances this instance refers to in a given state
  neighbors: Instance -> State,
  valuations: State -> LocalProperty -> Value }
{   all s: State | {
    State.valuations.Value = type.properties & LocalProperty
    this.defined[s]
     -- those instances which this instance refers to
    neighbors.s = {x: Instance | some p: type.properties & LocalProperty|
      s.valuations[p] = x } }}
one sig MainInstance extends Instance {}{ type = Main }
fact { one x: Instance | x.type = Main }-- there is only one Main Instance
-- value of local property p in state s
fun Instance::val [p: LocalProperty, s: State]: Value {
  p.(this.valuations[s]) }
fun Instance::queryVal[q:QueryProperty,s:State,ps:ParameterSetting]:Value{
  this.exprVal[q.expr,s,ps].val }
fun Instance::exprVal[ex:Expression,s:State,ps:ParameterSetting]:ExpressionValue{
 { ev: ExpressionValue| ev.expr = ex  && ev.@s = s && ev.y = this && ev.@ps.sameAs[ps]} }
pred Instance::defined[s: State] {
  -- each property has exactly one value in this state
  all p: this.type.properties & LocalProperty| one this.val[p,s] }
pred Instance::defaultVal[s: State] {
   -- by default all properties are set to null in a given state
  all p: this.type.properties  & LocalProperty| this.val[p,s] = DefaultValues.value[this.type] }
pred Instance::reachable[s: State]{
  -- only those instances reachable from the main instance are relevant in a given state
  this in MainInstance.*(neighbors.s) }

sig Value{}
sig NullValue extends Value { }
sig DefaultValues { value: Type ->one Value }
{ all t: Type| t in Model + Interface => value[t] = NullValue }

sig ParameterSetting { paramVals: Parameter -> lone Value }
fun ParameterSetting::params: set Parameter { this.paramVals.Value }
fun ParameterSetting::val[p: Parameter]: Value {  this.paramVals[p] }
pred ParameterSetting::agreesWith[ex: Expression]{  this.params = ex.usesParams }
pred ParameterSetting::sameAs[ps:ParameterSetting] {
  this.params = ps.params
  all p: this.params | this.paramVals[p] = ps.paramVals[p]  }

-- expression value determined by centered state and parameter setting
sig ExpressionValue  {
  expr: Expression,
  s: State,
  y: Instance,
  ps: ParameterSetting, -- parameter setting
```

```
57    val:  Value -- value for this expression, centered state and parameter values
58  } { ps.params = expr.usesParams
59    all ev: ExpressionValue| {ev!= this =>
60      { ev.@expr != expr || ev.@s!= s || ev.@y != y || !ev.@ps.sameAs[ps]  } } }
61
62  sig InstanceEvent { -- contains everything that determines the effect of an event e on a state
63  -- succ denotes all instance events directly triggered by this instance event
64    e: Event,
65    x: Instance,
66    v: ParameterSetting,
67    succ: InstanceEvent -> State
68  }{  let exprs = e.impactEdges.expr + e.edges.mappings.expr |
69      all s: State | all ex:exprs | one this.exprVal[s,ex]
70    e.~events = x.type -- Model that is type of Instance contains e
71    v.params = e.params
72    all s: State | {
73      {all j: succ.s|some g: e.edges| j.successorOf[this,s,g]} &&
74      { all g: e.edges  | let v = x.val[g.link,s]|
75        v != NullValue => some j: succ.s| j.successorOf[this,s,g]}}}
76  -- predicate that expresses that "this" instance event is triggered by i in state s
77  -- either using a push edge or a pull edge
78  pred InstanceEvent::successorOf [i: InstanceEvent, s: State, g: EventEdge] {
79    all p: this.e.params | let ex = g.paramExpr[p] |
80      this.valParam[p] = (i.exprVal[s, ex]).val
81    g in PushEdge => (g in i.e.edges &&
82      this.e = this.x.type.concreteEvent[g.target] && this.x = (i.x).val[g.link,s])
83    g in PullEdge => (g in this.e.edges && i.e = i.x.type.concreteEvent[g.target] &&
84      i.x = (this.x).val[g.link,s]) }
85  fun InstanceEvent::valParam[p:Parameter]: Value { this.v.val[p] }
86  -- all instance events reachable from this instance event in state s
87  fun InstanceEvent::scope[s:State]: set InstanceEvent {  this.*(succ.s) }
88  -- expression value of ex  in state s on this instance event
89  fun InstanceEvent::exprVal[ s: State, ex: Expression]: ExpressionValue {
90   { ev: ExpressionValue| ev.expr = ex  && ev.@s = s && ev.y = this.x && ev.ps.sameAs[this.v] }}
91
92  -- when we trigger instance event i in state s1, we end up in state s2
93  pred step[s1: State, i: InstanceEvent,  s2: State] {
94    i.x.reachable[s1] -- instance on which event is occurring is reachable in state s1
95    let scope = i.scope[s1] | {
96      -- iscope comprises all instance events in the scope associated with instance y
97      all y: scope.x | let iScope = (y.~x) & scope | {
98        all p: y.type.properties & LocalProperty| { p not in iScope.e.impactEdges.target
99          => y.val[p,s2] = y.val[p,s1]}
100       all j: iScope | all g: j.e.impactEdges |
101           y.val[g.target,s2] = j.exprVal[s1, g.expr].val }
102     all y: Instance-scope.x | all p: y.type.properties & LocalProperty |
103       y.val[p,s2] = y.val[p,s1] }}
104 pred show {} run show
```

Chapter 6

Discussion

We now compare traditional approaches presented in Chapter 3 and Chapter 4 to the Alloy-based approach described in the last chapter. 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. One 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 meta-modeling 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 meta-modeling 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 Section 5.2 in terms of the meta-model.

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, which again 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., meta-modeling 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 meta-model 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 imple-

mentation 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. For example 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.

Chapter 7

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 [17]). 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 formalized the EP language using both traditional approaches and a novel Alloy-based approach. 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.

Bibliography

[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 (ICSOFT'06)*, pages 63–71, Setúbal, Portugal, 2006.

[7] P. Kelsen. A declarative executable model for object-based systems based on functional decomposition. Technical Report TR-LASSY-06-06, Laboratory for Advanced Software Systems, University of Luxembourg, May 2006.

[8] P. Kelsen, E. Pulvermueller, and C. Glodt. Specifying executable platform-independent models using OCL. In *the Proceedings of Workshop Ocl4All: Modelling Systems with OCL*, ECEASST 2008(9), 2007.

[9] P. D. Mosses. Theory and practice of action semantics. In *the Proceedings of 21st Int. Symp. on Mathematical Foundations of Computer Science (MFCS'96)*, volume LNCS 1113, 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 (PSI'01)*, pages 165–190, 2001.

[11] OMG. Meta object facility (mof) core specification, v 2.0, Jan 2006.

[12] OMG. Object Constraint Language version 2.0, May 2006.

[13] OMG. Omg unified modeling language (omg uml) infrastructure v 2.1.2, Nov 2007.

[14] OMG. Unified modeling language superstructure specification, v 2.1.2, November 2007.

[15] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[16] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993.

[17] Y. Zhang and B. Xu. A survey of semantic description frameworks for programming languages. *SIGPLAN Not.*, 39(3):14–30, 2004.