Action Semantics applied to Model Driven Engineering

Gijs Stuurman

Master Thesis

Computer Science, Track Software Engineering

26 November 2010

Supervisors:
Dr. I. Kurtev
Dr. C. Bockisch
Dr. Selim Ciraçi

Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente, Enschede, The Netherlands

Abstract

Model Driven Engineering (MDE) is a software engineering approach using a high level of abstraction to model software systems. These models drive the development activities. The power of such models is that they are technology independent and are often expressed in Domain Specific Languages (DSL). The practice of MDE is to create a modeling language or metamodel in which to express models.

A language requires a formal definition of its semantics. For modeling languages there are various approaches to specify the static semantics. However, there is no standard approach to specify the dynamic semantics, which specifies the execution behavior of models.

Action semantics (AS) is a framework for the formal description of programming languages. Peter Mosses developed action semantics as a new formalism that would improve the shortcomings of denotational and structural operational semantics. AS aims to be both precise and pragmatic. A semantic framework needs to be precise by being unambiguous, complete and consistent. Its pragmatic aim is to aid the language creator by using a human readable, English-like syntax and including first class constructs to model common computational concepts such as control flow, bindings and storage.

In this work, action semantics is adapted from its origin in programming languages to modeling languages. Whereas programming languages use grammars and abstract syntax trees, modeling languages use metamodels and models, which may contain cycles. This difference is the biggest challenge that has been resolved.

This work provides action semantics as a semantic framework for modeling languages. This is done by modeling action semantics for MDE as a modeling language itself with the action semantics metamodel. Tool support consists of a compiler which compiles a model to an action tree that defines the semantics of a model. A simulator is also provided that can run action trees, either with a GUI or on the command line. This simulator allows the semantic description of a modeling language to be used for prototyping. Action semantics for MDE has been successfully used to specify the execution behavior of two modeling languages.

Acknowledgements

I would like to thank Ivan Kurtev for being my supervisor. Ivan provided guidance and feedback throughout this work, whilst illustrating the larger field of academic research. The friendly cooperation has been invaluable to realize this work.
I would also like to thank the other committee members; Christoph Bockisch and Selim Ciraçi for reviewing my thesis.

Gijs Stuurman, November 2010, Enschede

Table of Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**
1.1 Background 1
1.2 Problem Statement 2
1.3 Research Questions 3
1.4 Contributions 3
1.5 Outline 4

2 Basic Concepts **5**
2.1 Introduction 5
2.2 Model Driven Engineering 5
2.3 Semantics in general 8
2.4 Semantics for MDE 9
2.5 Semantic formalisms 10
2.6 Conclusion 11

3 Action Semantics **13**
3.1 Introduction 13
3.2 Origin 13
3.3 Example 14
3.4 Actions, Data and Yielders 16
3.5 State 17
3.6 Combinators 18
3.7 Facets 20
3.8 Theory of Action Semantics 20
3.9 Evolution of Action Semantics 21
3.10 Conclusion 21

4 Action Semantics for MDE **23**
4.1 Introduction 23
4.2 Modeling Action Semantics 23
4.3 Action Semantics metamodel 24
4.4 Tools and artifacts 28
4.4.1 Implementation of the Compiler 29
4.4.2 Implementation of the Simulator 31
4.5 Changes to Action Semantics 32
4.6 Conclusion 32

5 Challenges in applying Action Semantics to MDE **35**
5.1 Introduction 35
5.2 Denotations over models 35
5.3 Cycles 39
5.3.1 Unfolding and unfold 40
5.3.2 Problems with cycles in models 41
5.3.3 Compiler algorithm 42

5.3.4 Extension to the unfolding and unfold actions 44
5.4 Multiplicities 45
5.5 Inheritance 49
5.6 Conclusion 51

6 Case Studies **53**
6.1 Introduction 53
6.2 Activity Diagram 53
6.3 Production Line Language 60
6.3.1 Possible improvements 64
6.4 Conclusion 66

7 Conclusion **67**
7.1 Introduction 67
7.2 Summary 67
7.3 Evaluation 68
7.4 Comparison to Related Work 70
7.5 Discussion 72
7.6 Future Work 75

References **77**

Appendix A: Action Semantic Descriptions **81**

Appendix B: Action Trees **99**

Appendix C: CD **103**

1 Introduction

1.1 Background

Instead of using a programming language to solve a problem, one can create its own modeling language to solve problems in a domain. Model Driven Engineering (MDE) is a software engineering approach to create modeling languages using a high level of abstraction to model domains [Ken02]. The practice of MDE is to create a modeling language or a metamodel in which to express models. The power of such models is that they are technology independent and are often expressed in Domain Specific Languages (DSL) [Deu00]. The high level of abstraction in MDE is high in relation to the distance to the implementation of software systems or programming languages. Using this high level of abstraction the models can be expressed in terms of the domain, making it more expressive and better suited for use by domain experts.
There are multiple frameworks to define the abstract syntax or structure of a modeling language. An open problem is the definition of semantics for modeling languages. Semantics formally define the meaning of a language and programs, model or sentences expressed in that language. The semantics of a language allows reasoning, comparison and analysis. This holds for any language from mathematical notation, natural language, such as English, to programming languages. These uses require a semantic formalism to be precise, while usability of a semantic framework requires it to be pragmatic.
Static semantics define the well-formedness, typing and structure of a sentence, program or model. For programming and modeling languages the dynamic semantics define the computational behavior of a program or model. Dynamic semantics define the run-time behavior during the execution of a program or model. As the purpose of most modeling languages is to ultimately create executable software, a formalism to define their dynamic semantics is required.
Different semantic formalisms have already been explored for adaptation for MDE. Most of these formalism originated in the field of programming languages. These include structural operational semantics for MDE [Wol09], graph transformations [Lar06] and others [Rom07,Che05]. We propose to explore action semantics [Mos92] for defining the semantics of modeling languages.
Action semantics was created as a semantic formalism to define the dynamic semantics of programming languages by Peter Mosses [Mos92]. Mosses aimed to produce a semantic framework that was as precise as existing frameworks, but was also more pragmatic by including constructs for ordinary computational concepts such as control flow, bindings and storage. By using a human readable,

English like syntax instead of a mathematical notation action semantic is also more pragmatic by being easier to read, understand and define. Action semantic specifications are also aimed to be composable, which fits the possibility of composing modeling languages.

1.2 Problem Statement

MDE is the practice of creating modeling languages with metamodels with which one can express models. Because MDE is a software engineering approach most of these models become software artifacts that can be executed. The behavior of a model is the execution as defined by its dynamic semantics. Thus an approach is needed to specify the dynamic semantics of models in MDE.

Current approaches such as using informal specifications in natural language as found in design documents are insufficient for these purposes as these can be ambiguous and imprecise. Another way is to formally specify semantics by mapping the semantics onto a well understood domain. Existing approaches to specify semantics map onto mathematical concepts, such as the lambda calculus, logic or graph transformations. Most of these formal approaches were first applied to programming languages.

Formal specification approaches might be sound but suffer from the large conceptual gap between the domain of the model and the mathematical constructs. This makes the semantics difficult to understand, specify and to use, which is particularly caused by the difficulty of grasping and defining the large mathematical equations that are required.

Therefore a way to specify the semantics for modeling languages is needed that is formal and usable. The required formal qualities are to be unambiguous, complete and consistent to support reasoning, code generation, verification and validation. The usability property is defined as human readable, modular, composable and matching the level of abstraction in MDE to aid the designer of a modeling language.

Action semantics [Mos96] occupies the middle of the gap between a domain and mathematical constructs. The specifications are in a readable and English like syntax, modular and composable to ease the construction and usage of the semantics. The level of abstraction is higher than mathematical approaches by introducing common computational concepts, such as storage and abstractions. The aim of action semantics is to produce formal semantic specifications that reflect ordinary computational concepts and are easy to read, understand and compose. Thus a formalization framework that is both precise and pragmatic.

Action semantics originates from the field of specifying semantics for programming languages. It has been used to specify the semantic definitions for Pascal, ADA, Java and ML [Wat09]. Because of its origin with programming languages action semantics is designed to work with grammars and abstract syntax trees. In MDE metamodels and models serve the same purpose but are different by being object-oriented and being graphs rather than trees.

The aim of this thesis is to explore if action semantics can be applied to MDE to provide formal, understandable, modular and usable semantic specification of models.

1.3 Research Questions

The research objective of this thesis is to adapt action semantics to specify dynamic semantics for modeling languages. The following research questions will guide the adaptation of action semantics from the field of programming languages to MDE.

- RQ 1: Can action semantics be applied to modeling languages in model driven engineering, having its origin from programming languages?
  - RQ 1.1: Model driven engineering languages represent models as graphs, programming languages use abstract syntax trees. What adaptations are needed for using graph structures?
  - RQ 1.2: Programming languages use grammars whereas model driven engineering uses metamodels. How can action semantics be used with inheritance, attributes, references and multiplicities as used in metamodels?
- RQ 2: Action semantics is a good fit for programming languages because the entities map more closely to computational constructs than the mathematical approaches. Which level of abstraction of computational constructs in action semantics is needed for model driven engineering?
- RQ 3: What tool support is needed to allow action semantics to be used in model driven engineering?
  - RQ 3.1: How can the behavior or dynamic semantics of a model be shown by using action semantics and a tool?

Questions related to the composability of semantic specifications for different metamodels and the composition of metamodels are not explored. We do not explore if the human readability of AS and its other pragmatic qualities bring benefits to developers.

1.4 Contributions

This work provides the following contributions:

- Adaptation of Action Semantics from programming languages to MDE

  In this work, action semantics is adapted from its origin in programming languages to modeling languages. Whereas programming languages use grammars and abstract syntax trees, modeling languages use metamodels and models, which may contain cycles. This difference is the biggest challenge that has been resolved.

- Action semantics metamodel

  Action semantics can be used to specify the dynamic semantics of modeling languages. For this purpose action semantics itself is modeled as a modeling language. Chapter 4 introduces the action semantics metamodel and the tool chain for using action semantics for modeling languages. Chapter 5 addresses the design of the metamodel by describing the challenges that have been met.

- Tool support: Compiler

  Given the action semantic specification of a modeling language and a model conforming to that language, the compiler compiles the semantic specification of the execution behavior of the model. This specification is in the form of an action tree, which conforms to the action semantics metamodel. Section 4.4 and 5.3 discusses the internal working of the compiler.

- Tool support: Simulator

  The simulator provides an implementation of action semantics that can be used to execute action trees. The simulator provides both a GUI that supports stepping through an action tree and a command line interface. With the simulator the semantic specifications of models can be prototyped.

1.5 Outline

This thesis has a linear structure. Readers familiar with the basic concepts in chapter 2 or action semantics in chapter 3 may skip those chapters.

Chapter 2 introduces the basic concepts. First model driven engineering (MDE) is introduced. Then semantics in general, semantics for MDE and semantic formalisms are discussed.
Chapter 3 explains action semantics. After the introduction of the origin and motivation for action semantics, the formalism is introduced with a simple example. Then the constructs from action semantics are explained. Finally the theory and evolution of action semantics are discussed.
Chapter 4 lays out the approach for using action semantics for MDE. Action semantics itself is modeled as a modeling language with the action semantics metamodel. Then the tools and artifacts that are used when defining the semantics of a modeling language are described.
Chapter 5 documents how the challenges that have been encountered have been met. Most of these challenges have to do with the differences between the use of grammars and abstract syntax trees in programming languages, while modeling languages use metamodels and models, which are graphs. The biggest challenge of possible cycles in graphs that model looping behavior is addressed in section 5.3.
Chapter 6 shows two case studies of using action semantics to specify the dynamic semantics of modeling languages with action semantics. The first case study is a modeling language with possible cycles in its model and illustrates the challenge from section 5.3. The second case study is a domain specific modeling language with semantics on a higher level of abstraction than seen in earlier examples.
Chapter 7 concludes this work with a comparison to related work, an evaluation and a discussion. Finally, future work contains possible continuations upon this work.

2
Basic Concepts

2.1 Introduction

This thesis introduces the use of action semantics to specify the execution semantics for modeling languages built using model driven engineering. First model driven engineering is explained. Then semantics in general and its application in the field of model driven engineering are introduced. Semantic formalizations other than action semantics close out this chapter. Action semantics is explained in depth in the next chapter.

2.2 Model Driven Engineering

Model driven engineering (MDE) provides a higher level of abstraction for software engineering by using models as abstractions. In MDE, models are the unification concept in the same way that objects are in object-oriented programming. Model driven architecture is an approach to software system specification that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform [OMG03]. MDE extends model driven architecture (MDA) in which models are the principle artifacts, with the notion of the software development process [Ken02]. MDA practices and standards are defined by the Object Management Group (OMG), a consortium consisting of researchers and industry. OMG promotes the use of modeling and models as the main activity and artifacts of software development. The use of models directs the course of understanding, design, construction, deployment, maintenance, interoperability and modification [Mil03].
With models being the most important concept in MDE a proper definition of a model is needed; *A model is an abstraction of a part of reality and is expressed in a modeling language. A model provides knowledge for a certain purpose* [Kur05]. This definition conveys that models abstract from reality and help understanding, communication and and analysis. This is a common practice in various engineering disciplines of separating a specification from its implementation.
The level of abstraction in model driven engineering purposefully creates a gap between the domain and the implementation technology. This gap allows for technology independence and portability of software systems. This has been a goal of OMG, particularly after the quick introduction of various technologies that use the web.
The practice of MDE, therefore, is to create modeling languages that can express

models. This modeling language is defined as a metamodel: *A meta-model is a model of the conceptual foundation of a language, consisting of a set of basic concepts, and a set of rules determining the set of possible models denotable in that language* [Fal98].
A model expressed in a modeling language has a “conformsTo” relation between the model and its meta-model. This “conformsTo” relation is similar to the “instanceOf” relation in object-oriented programming and the way that a sentence in English is part of the English language. A metamodel defines the abstract syntax of a language.
Domain Specific Languages (DSL) are often used in combination with model driven engineering. Domain specific languages are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application [Mer05]. Models can be built with domain specific languages as they allow for more expressiveness and better usability by using a specific syntax. By transforming models expressed in DSLs to an intermediate representation conforming to a metamodel, the technology and implementation independence remains. This is in contrast to embedded DSLs that commit to an implementation or programming language.
Metamodels itself need to be expressed in some modeling language as well. We use the standard metamodel called Ecore [Eco]. Ecore is similar to the MetaObject Facility (MOF) provided by OMG and is also similar to UML. The Ecore standard is part of a larger technology set called the Eclipse Modeling Framework (EMF), that includes support for code generation and creation and editing and serialization of models. As is common with other languages such as BNF the top level language is expressed in itself, to avoid infinite recursion.
Following [OMG03] the relation between the different models is as follows:

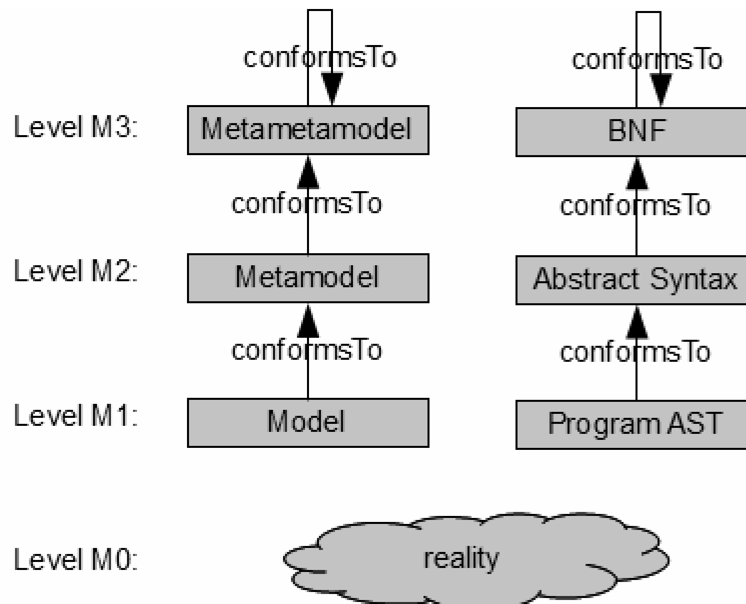


Figure 2.2.1: Model levels

Of course there is also a relationship between the model level M0 and M1, as the models are always designed for a certain purpose and model a part of reality. The relationship between reality and a model is sometimes call "representedBy" [Bez05]. This relationship is omitted as it is of a much less formal nature than the relationships between the model artifacts. The inverse of the conforms to relation is called the "modelOf" relation.
With the reality and Ecore metametamodel being fixed, most of the design activities in MDE concern the creation of a metamodel in which models can be expressed.
Most of the semantic formalizations that will be explained or used in this thesis have their origin with programming languages. Therefore a comparison between programming languages and MDE is presented here.

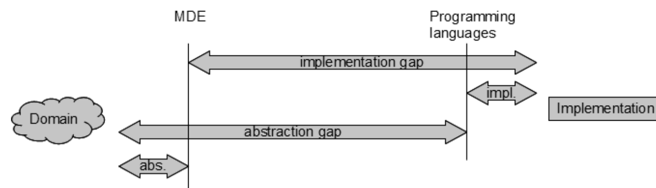


Figure 2.2.2: MDE and programming languages

Fig 2.2.2 depicts the different positions of MDE and programming languages with regard to the gap between a problem in a domain and the solution in the form of the implementation of a system. The higher level of abstraction in MDE creates a larger gap towards the implementation. This is the technology independence that MDE aims for. The implementation gap between MDE and an implementation is bridged by transforming models into other model or code which finally results in an implementation.
Another difference between MDE and programming languages is the representation of an instance of a modeling or programming language. For a programming language this is a program, in MDE this is a model. A program is represented by an abstract syntax tree. This tree is the result of parsing a program written in a concrete syntax. This concrete syntax is often textual, but might as well be visual. An instance in a modeling language is a model, which is a graph. A representation of a model when using Ecore is an XML file. This means that there is a more clearer distinction between the concrete syntax and an abstract representation in MDE than there is in programming languages.

	Programming languages	**Model Driven Engineering**
Focus	General purpose	Domain specific
Defined by	Grammar	Metamodel
Instances	Abstract syntax tree	Model (graph)

Table 2.2.1: Comparison between MDE and programming languages

Table 2.2.3 sums up the differences between programming languages and MDE. Particularly the representation of instances of a language with graphs rather than trees will be a challenge for adaptation of action semantics for MDE, as will be shown in later chapters.

2.3 Semantics in general

Semantics formalize the meaning of things. Knowing the meaning of something is required to be able to reason about, compare and analyze it. The most common way to communicate the meaning of something is to use natural language such as English. But natural languages can be quite ambiguous and imprecise. A more formal way to communicate meaning is through mathematical constructs. But even mathematical notation requires a specification for its semantics. For instance the expression $3 * 2 + 5$ is only correctly interpreted when the precedence rules for multiplication and addition are correctly applied. Thus the semantics for mathematics gives the unambiguous interpretation of $(3 * 2) + 5 = 6 + 5 = 11$. When the semantics is not formally defined someone needs to make his own interpretation which may lead to different interpretations.
Therefore there is a need to be able to specify semantics unambiguously, completely and consistently. To be unambiguous is to be understood in only one way. This means that a semantic specification should be interpreted the same by everybody. Semantic specifications are complete when the meaning for all constructs or compositions are expressed or can be derived from the specifications. A complete semantic specification may however include parts that are explicitly not specified. For instance the order of evaluation. Finally, a semantic specification should be consistent, such that parts of the specification will not lead to contradictions.
These qualities have to be met by both the language or domain with which semantics are expressed as well as for the specifications that are made with this language. Of course the former should support and guide the latter. When semantic specifications are unambiguous, complete and consistent, they can be used with mathematical techniques such as validation, proofs and simulation [Mos92].

The semantics of programming languages and modeling languages serve these purposes:

- *Reasoning and documentation*: the semantics specification documents the design and meaning of models which is needed to be able to communicate between users of a modeling language
- *Implementation*: a designer needs the semantics in order to build an implementation
- *Code generation*: tools that generate code for an implementation need the semantics to do so
- *Verification and validation*: the semantics are needed to be able to check whether models conform to certain specifications

For programming and modeling languages the semantics can be divided into static and dynamic semantics:

- *Static semantics*: Semantics dealing with the structure of a construct, model or program such as its well-formedness and typing
- *Dynamic semantics*: Semantics describing the behavior or run-time effect of a construct, model or program

This thesis uses action semantics as a semantic framework to specify the dynamic semantics of modeling languages. Therefore, unless specifically noted otherwise, we mean dynamic semantics when referring to semantics.

2.4 Semantics for MDE

This section explains semantics in the context of MDE. For modeling languages the semantics serve various different purposes, as they do for programming languages. The same distinction between static and dynamic semantics from programming languages can also be made. Finally, the design process of modeling languages and the artifacts in MDE create requirements for a semantic framework to be useful with MDE.

Semantics for MDE serves the same purposes as for programming languages that were explained in the previous section. These purposes are reasoning and documentation, implementation, code generation and verification and validation. Another usage of a semantic specification is to create a prototype from it. This is possible for programming languages as well, but fits particularly well with the philosophy of MDE to create implementation technology independent metamodels. Building a prototype from a semantic specification is related to using it for the implementation and code generation purposes. It can also be a useful aid for the reasoning and documentation and verification and validation. With prototyping a working system can be created for a model conforming to a metamodel before the transformations into a specific technology platform need to be defined.

The distinction between static and dynamic semantics from programming languages is also applicable to MDE:

- *Static semantics*: Well-formedness and typing of metamodels and models and the conforms to relation between a model and its metamodel
- *Dynamic semantics*: Semantics describing the execution behavior or run-time effect of a model

Static semantics specify the allowed structures in a language, such as well-formedness and proper typing of models. For natural and programming languages the static semantics are expressed in the grammar of the language. In MDE metamodels are used instead of grammars. Which models are allowed is defined by the structure of their metamodel. Where this object-oriented structure of a metamodel is insufficient in expressing constraints on models, annotations can be used. A common language used for expressing constraints is the Object Constraint Language [OCL03]. For instance a constraint concerning two attributes can not be expressed in the object-oriented structure and requires a separate annotation. This is the same as with languages where constraints that can not be expressed in the grammar are described in an additional notation.

The dynamic semantics for a modeling language is the computational meaning of model. This is the execution behavior or run-time effect of a model. This computational meaning specifies the behavior of a model during an execution and the end result if any. Through the “conformsTo” relation both the static and dynamic semantics for models follow from their metamodel. This is the same in natural and programming languages where the semantics for a language specify the meaning of a sentence, program or model expressed in that language.

Creating modeling languages is the primary activity in MDE. For each new language that is created the semantics for the language need to be defined. How to express the static semantics for this language follows from the choice of using an MDE approach and the Ecore metamodel. However, currently there is no standard approach within MDE to express the dynamic semantics.
For programming languages the semantics are specified for grammars and can be applied to programs in that language. For modeling languages the semantics need to be defined for a metamodel and this specification defines the semantics for a model conforming to that metamodel. A semantic framework for MDE should therefore work nicely with the constructs used in metamodels.
Model driven engineering is a software engineering approach that shares the characteristics of other engineering approaches of being an iterative process. Therefore the creation of models and software is usually an evolution. A semantic framework for specifying the semantics of modeling languages should fit nicely with this process, where the semantic definition of a language can follow the evolution of the language. Also in MDE and modeling in general, abstraction is a major part of the design effort as well as striving for composability. An approach to specifying semantics should also share these characteristics.

2.5 Semantic formalisms

The meaning of a language is defined by mapping the structure of a language onto a semantic domain [Har04]. The structure of a language is the abstract syntax following the grammar of a programming languages or the metamodel of a modeling language. A semantic domain for the purpose of expressing computational behavior can be anything from a description in natural language to a more precise mathematical domain. Possible mathematical domains range from logic reasoning to computational descriptions using the lambda calculus. The following are popular ways for providing this mapping and domain: Axiomatic Semantics, Denotational Semantics, Structural Operational Semantics, Graph Transformations and Action Semantics.

- *Axiomatic Semantics*
  Axiomatic semantics is based on Hoare triplets and logic [Hoa69]. Given a part of the language and a matching precondition, the post condition will hold. Composing these axioms will create a deduction tree for the whole semantic specification of program or model.

- *Denotational Semantics*
  Denotational semantics uses semantic equations over parts of a language to map onto mathematical objects [Sto77]. The semantic functions are applied on the root of an abstract syntax tree that conforms to the grammar of the language. As semantic functions may apply semantic functions to sub parts this recursive application over an entire abstract syntax tree gives one specification for a program. The mathematical objects are functions from the state before into a new state that capture the effects of statements in a language. The whole specification for a program is function from the input state to its output. Various mathematical constructs have been used as a domain in denotational semantics. Among these was the lambda calculus. Later monads were used to represent state and for

composability of semantic specifications. See for instance the aptly named "Semantic Lego" thesis about modular denotational semantics. [Esp95].

- *Structural Operational Semantics*
  Structural operational semantics is also a compositional formalization based on labeled transition systems [Plo81]. Compositional means that the semantics for a structure is defined by the composition of the parts of that structure. In the transition system the nodes represent the run-time state of a program. Rules based on the structure of a language are used to construct proof trees. Each of such proofs is a transition to a new state. In [Wol09] Structural Operational Semantics is applied to MDE.

- *Graph Transformations*
  Graph transformations [Bar02, Hec06] also define a labeled transition system. The nodes are graph representations of a state of a program. The transitions are graph transformation rules that match and alter the graph.

- *Action Semantics*
  Action semantics [Mos92] uses the mapping constructs from denotational semantics to map onto a domain that has first class entities for common computational constructs such as control flows, bindings and storage. Action semantics is explained in depth in chapter 3.

2.6 Conclusion

This chapter explained the concepts of model driven engineering and semantics. The usages of semantics for MDE and semantic formalizations were introduced. The rest of this thesis will show action semantics as a framework to specify the semantics for modeling languages created with MDE. The basic concept not discussed so far is action semantics itself, which will be introduced in the next chapter.

3 Action Semantics

3.1 Introduction

This chapter introduces the final basic concept, which is the semantic formalism action semantics. Action semantics and its origin are introduced in section 3.2. To aid the understanding of action semantics for unfamiliar readers, this chapter provides a simple calculator language as a running example. This example is explained in section 3.3 that also introduces the concepts which action semantics shares with denotational semantics. The main concepts of action semantics of actions, data and yielders are introduced in section 3.4. Section 3.5 explains the modeling of state through computations. Section 3.6 introduces action combinators that allow new actions to be defined by combining actions. Action semantics uses an organization into facets explained in section 3.7. This chapter closes with the theoretical background and evolution of action semantics in section 3.8 and 3.9.

3.2 Origin

Action semantics (AS) is a framework for the formal description of programming languages [Mos92]. Peter Mosses started developing action semantics in 1992 as a new formalism that would improve the shortcomings of denotational and structural operational semantics. Mosses found making formal specification with denotational semantics difficult to create accurately, to modify and to extend, despite its elegant theory. Basic computational concepts such as control flows, bindings and storage get obscured by the mathematical notation in denotational semantics [Slo95]. The aim of action semantics is to produce formal semantic specifications that reflect ordinary computational concepts and are easy to read, understand and compose. Thus a formalization framework that is both precise and pragmatic.
Action semantics has first class entities for control flows, bindings, modification of state and parameter passing. By providing a notation and a way to compose these concepts, these are not obscured by an encoding into a mathematical notation such as the lambda calculus. Mosses also created a new notation to use with these concepts. In the earliest versions of action semantics these were combinator symbols such as $\oplus$ and $\otimes$. As this notation suffered from the same readability issues as the lambda calculus, Mosses opted for a human readable, English like syntax. This syntax for action semantics is called “action notation”.

Action semantics follows the approach of denotational semantics where syntactic entities on the left hand side of the semantic equation are mapped onto its denotation on the right hand side. These semantic equations or functions make the denotational specifications composable. However the semantic functions do not map onto a mathematical domain as with denotation semantics but onto actions. Actions are written in action notation. This action notation has the first class entities actions, data and yielders to describe the concepts of control flow, bindings and state. Denotational semantics allows for arguments in the semantic equation that are passed into the definition on the right hand side. In action semantics there is no such argument as an action always is a function from an input state to an output state. Semantic equations in Action Semantic Definitions consist of a name, a part to which the equation is applicable and the action notation on the right hand side. For readability Mosses introduced a notation that mixes prefix, infix notation of actions together with the usage of a vertical bar to group actions. Nested action form a tree, which in this notation has its root node on the left and uses vertical bars to indicate branching.

3.3 Example

This chapter has a running example to illustrate various parts of action semantics. The example is a simple language that models calculations on a calculator. This language includes simple arithmetic and a way to store and recall a number from memory. This memory function is usually present on a simple calculator with the M^+ button for storing a number by adding it to the current number in memory and the M^{rec} button to recall the number. The example is based on the calculator example from [Slo95].

```
Calculation ::= Expr+

Expr ::= Expr "+" Expr
       | Expr "M+"
       | "Mrec"
       | Number

Number ::= "0" .. "9"
```

Figure 3.3.1: Grammar for the calculator language

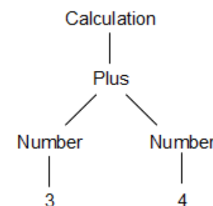


Figure 3.3.2: Abstract syntax tree of a simple addition

Fig 3.3.2 shows an abstract syntax tree representation of a calculation conforming to the grammar of the calculator language shown in Fig 3.3.1. The concrete syntax or transformation to arrive at the abstract syntax tree by pressing buttons on a calculator is not shown here.
To get a semantic specification of the simple addition example the semantics for the calculator language will be defined using action semantics. This specification is a collection of semantic equations similar to an approach using denotational semantics. The application of these semantics equations to the abstract syntax tree produces the semantic specification of the execution of the simple addition.

```
01: evaluate Plus [lhs:Expr rhs:Expr] =
02: | | evaluate $lhs
03: | and
04: | | evaluate $rhs
05: then
06: | give (sum (given integer-sort 1, given integer-sort 2))

07: evaluate Number [n:Int] =
08:   give yield $n
```

Figure 3.3.3: Semantic equations for the calculator language

Fig 3.3.3 shows the two semantic equations corresponding to the *Expr* + *Expr* and *Number* parts of the grammar. On the left hand side of the equation is its name and the part of the grammar the equation can be applied to. On the right hand side is the action notation that describes the semantics for that part. This right hand side includes other semantic equations applied to sub parts. This can be seen in line 02 and 04 in the semantic equation for *Plus* that applies semantic equations to its left hand side and right hand side. This composability that originated in denotational semantics allows the expression for Plus to be expressed without knowing exactly what the sub parts of the expression are. This matches very closely to the structure of the rules in the grammar and the general design principle of abstraction. The notation with the dollar sign is used to refer to parts on the left hand side of a semantic equation that are sub parts of an abstract syntax tree. The notation introduced by Mosses' uses capital letters for these references.

The application of the semantic functions to the abstract syntax tree of the addition example produces the action tree that is the semantics for execution of the expression 3 + 4:

```
01: | | give (yield 3)
02: | and
03: | | give (yield 4)
04: then
05: | give (sum (given integer-sort 1, given integer-sort 2))
```

Figure 3.3.4: Action tree representing the behavior of the simple addition

The end result of running the action tree from Fig 3.3.4 will produce the number 7.

This example showed various parts from action semantics: defining semantic equations that match the structure of the definition of a grammar, composability of semantic equations and the composability of actions. Hopefully this example was also intuitively understood as per Mosses' readability claims.
The example illustrated how action semantics can be applied to define the semantics of a language. The following section will explain the concepts from action notation and the action semantics model for computation in more detail.

3.4 Actions, Data and Yielders

An action represents computational behavior by information processing. Values passed to an action are used to generate new values that reflect changes in the state of the computation. Actions are steps in a computation that may complete (finish successfully), fail or diverge (not finish at all). Actions are functions but they are restricted in their allowed input and results to allow composition.
The information flowing through and between actions and stored in memory is data. This data is classified into algebraic sorts in action semantics. In action semantics there are sorts for the usual data such as integers, truth values, lists, sets and maps and new sorts can be defined. Sorts can be defined in action semantics by algebraic specifications that specify mathematical objects and their operations. Data specific to action semantics such as memory cells, bindings and abstractions are also defined as sorts. The data in action semantics can be qualified according to their lifetime and propagation between actions:

- *Transient*: intermediate results from actions
- *Scoped*: bindings of tokens to data
- *Stable*: data stored in cells in memory

Yielders are contained within actions and produce data. Their evaluation might use the information passed into action of any of the above mentioned types. The evaluation of a yielder cannot change any information as state mutation can only be done by actions.

To aid the understanding of the previous concepts the action and yielders that are used in the example from the calculator language are explained in more detail. In the action tree in Fig 3.3.4 the following actions and yielders were used:

- Actions
  - *give* Y: Gives the data yielded by evaluating the yielder Y
  - A_1 *and* A_2: Combines actions A_1 and A_2 by passing the incoming data to both and by merging the result from both actions. This is parallel composition of actions.
  - A_1 *then* A_2: Composes actions A_1 and A_2 by passing the incoming data to A_1 and the result thereof to A_2, which result is the result for the whole action. This is sequential composition of actions.
- Yielders
  - *yield Datum*: Puts *Datum* into the transient data
  - *given Sort* N: Retrieves the Nth datum from the transients and checks if the data belongs to *Sort*
  - *sum* Y_1 Y_2: Puts the result of the addition of Y_1 to Y_2 into the transient data

3.5 State

The state at run time of an action consists of transients, bindings and storage that hold data. Actions can modify this state and yielders can produce data from this state. The transients are a list of data that hold intermediate results to be passed between actions. This is information with the nature of use it or loose it. Any information that a subsequent action may need can be put into the transients by previous actions. The bindings consist of bindings from tokens to data. These bindings are identifiers for data or memory locations in the storage. The storage models the computational concept of memory and more specifically a heap. In action semantics the memory is abstracted into cells that have an index and can store any data sort.

The following semantic equations specify the semantics for the M^+ and M^{rec} button from the calculator language.

```
01: meaning Calculation [exprs:Expr*] =
02: | | allocate a cell
03: | then
04: | | | store (yield 0) in (given cell-sort 1)
05: | | and
06: | | | bind "memory" to (given cell-sort 1)
07: before
08: | evaluate $exprs

09: evaluate M+ [expr:Expr] =
10: | | evaluate $expr
11: | and
12: | | give (bound cell-sort to "memory")
13: then
14: | | store (sum (given integer-sort 1)
                (stored integer-sort in (given cell-sort 2)))
          in (given cell-sort 1)
15: | and
16: | | give (given integer 1)

17: evaluate Mrec [] =
18:   give (stored integer-sort in (bound cell-sort to "memory"))
```

Figure 3.5.1: Semantic equations for the memory construct of the calculator language

The calculator has one memory location into which a number can be stored and from which a number can be retrieved. The default number in this memory location is 0, this behavior is expressed in the semantic equation for *Calculation* in line 04 in Fig 3.5.1. In action semantics there is a requirement for a semantic function called "meaning" that can be applied to the root of a grammar. Here this root is *Calculation* and defines the initialization of the memory that is required in every semantic specification of a program in the calculator language.

The actions and yielders related to storage used in these semantic equations are:

- Actions
  - *allocate a cell*: This action both allocates a new cell in the storage and puts this cell in the transients as well. This is one of the few actions in action semantics that touches two types of data at the same time.
  - *store* Y_1 *in* Y_2: Stores the data produced by the yielder Y_1 in the cell yielded by Y_2
  - *bind* T *to* Y: Creates a binding from token T to the data produced by the yielder Y
- Yielders
  - *stored* S *in* Y: Gives the data stored in the cell yielded by Y when the data is of sort S
  - *bound* S *to* T: Gives the data bound to the token T when it is of sort S

Most of the attention so far has been on actions that either touch the transients, bindings or storage. The following section introduces the combinators that combine these actions.

3.6 Combinators

In action semantics every action has a precise goal and usually only touches either the transients, bindings or storage part of the state. This allows the action to be intuitively understood. To build more complex actions and behavior these small buildings blocks can be combined with combinator actions. Combinators combine two actions by directing the flow of information into its sub actions and by merging the results.
The semantic equation for a *Calculation* in the calculator language initializes a storage cell that will be the memory in the calculator. Part of this equation Fig 3.5.1 is repeated here. The *then* and *and* combinators are used here to combine different actions that all have a specific purpose.

```
01: | allocate a cell
02: then
03: | | store (yield 0) in (given cell-sort 1)
04: | and
05: | | bind "memory" to (given cell-sort 1)
```

Figure 3.6.1: Actions and combinators from the calculator language

As said the state in action semantics consists of transients, bindings and storage. The following figure shows how the flow of this state is handled by combinators throughout the execution of actions in Fig 3.6.1:

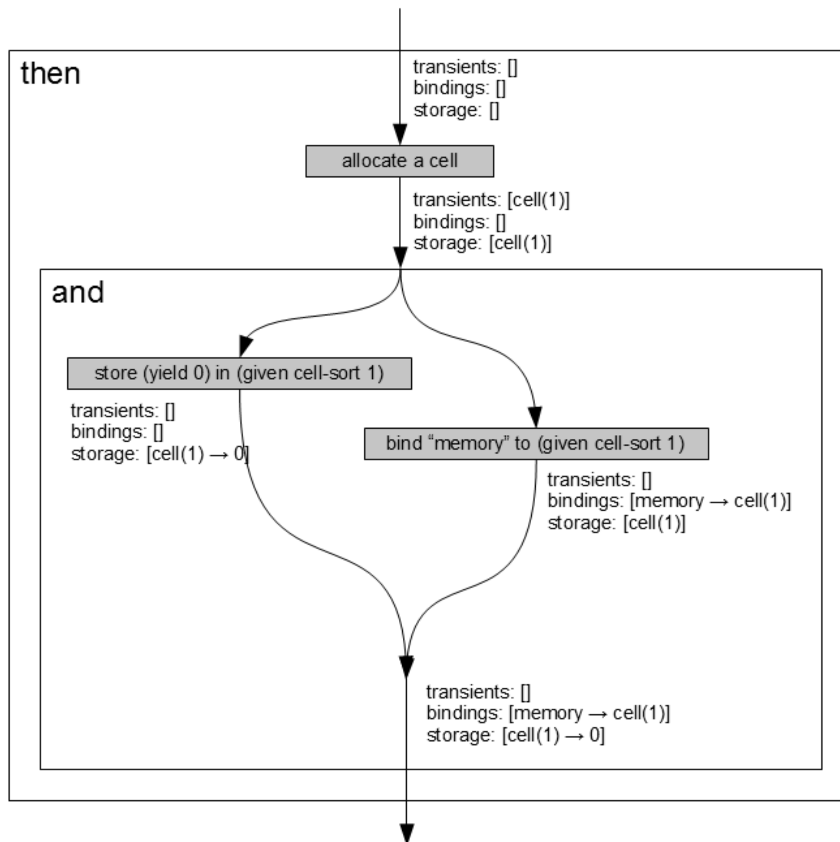


Figure 3.6.2: Flow of state as defined by the then and and combinator from Fig 3.6.1

The *then* combinator passes the output from one action into the next one. The *and* combinator combines two actions be giving them the same input and merging their output. After the *allocate a cell* action the transients consists of "cell(0)". Both the subsequent *store* and *bind* action require this data in the transients to store a number in the cell and to bind a token to it respectively. The *and* combinator does just that passing both the same input and combining their output. Also for both the *store* and *bind* action it holds that "cell(1)" is no longer in the transients after the action. This is exactly the definition of transients and their "use it or loose it" nature.
Combinators also have particular behavior when an action does not complete by producing a new state with transients, bindings and storage but fails instead. For both the *then* and *and* combinator when any of the sub actions fails, the whole combinator action will fail and have *Failure* as its result. For instance a *given Sort N* yielder can fail when there is no Nth data or when it is not of sort *Sort*.

Fine grained actions together with combinators allow for a host of different computational constructs to be expressed in action semantics. The use of com-

binators also works very well with the approach from denotational semantics where the semantics of a program is constructed from the building blocks of semantic equations. Combinators also allow sub actions to only concern themselves with the information from the state that they require, the flow of the other information is handled by combinators to supply a subsequent action with the information it may need.

3.7 Facets

Mosses categorizes actions according to the type of information they affect. Each of these categories is called a facet. A key notion is that a combinator behaves neutrally with the information from others facets. Therefore the flow of information from another facet is implicit in a combinator. This allows for the introduction of new facets that do not require an upgrade to existing facets or specifications using those facets. This is Mosses' approach to solve the problem from other semantic formalizations where the introduction of new information impacted every description by requiring a rewrite to encode the new construct. The following facets are defined in action semantics:

- Basic facet: actions that concern control flow such as choice, interleaving and sequencing
- Functional facet: actions and yielders that produce and consume transient data
- Declarative facet: actions and yielders that produce and consume bindings
- Imperative facet: actions and yielders that allocate, inspect and update the storage
- Reflective facet: actions and yielders that represent and enact abstractions of actions
- Communicative facet: actions and yielders that model agent based concurrency

There is also a hybrid facet for actions that use information from more than one facet at once. There are surprisingly few actions in the hybrid facet. This supports Mosses' modularity claim as most actions belong to a facet that affects only of type of one type of information [Wan97].

3.8 Theory of Action Semantics

Mosses started the development of action semantics by improving on the pragmatics of semantic formalizations. However a formalization is not usable without a well-founded theory. Action semantics is a denotational semantics that map onto the action notation that Mosses developed. This action notation has been formalized by using “Unified Algebras” [Mos89] and by mapping action notation onto “Structural Operational Semantics” [Plo81, Mos92] and later “Modular Structural Operational Semantics” [Mos99].

Unified algebra is used to define the data notation that are in action semantics such as integers, text, lists and maps. Also any data sort a user may want to define need to be defined using unified algebra. Unified algebra is also used to define action notation. For instance, for combinators the algebraic properties such as being associative, total or partial and/or commutative are defined. As well as the action that is the unit for a combinator action. Using these techniques Mosses defined a kernel of action semantics into which the whole action notation can be reduced. This kernel has been formalized using (modular) structural operational semantics. Mosses lauds his unified algebra as a nice fit for the formalization of action semantics. But in [Wat09] this choice for an unknown formalism is cited as a reason against adoption of action semantics. Theories for testing equivalence and bisimularity are also given by Mosses [Mos92].
In [Wan97] action semantics and "Modular Monadic Semantics" are synthesized into one formalism. Modular monadic semantics is used to formalize the semantics of action semantics. This approach also allows new facets to be formalized by building upon the semantic specification of the facets from the kernel of action semantics. Mosses had a similar aim by redefining the semantics with modular structural operational semantics [Wat09].

3.9 Evolution of Action Semantics

Through the years action semantics has been used to specify the semantics for various programming languages. There are action semantic descriptions for (parts of) Ada [Mos92], PASCAL, Java and ML done by Mosses together with Watt and Brown. The "Architecture and Language-Neutral Distribution Format" [Tof93] has been developed in industry and uses action semantics to formalize semantics. Also tools have been developed for compiler generation based on action semantic descriptions. These are the Actress [Bro92], ANI [Mou93], Albaco [Mou99], Cantor [Pal92] and OASIS [Orb94] projects. In [Doh93] action semantic descriptions are used to create prototypes. The usage of action semantics prompted Mosses to review the action notation which led to AN-2 [Las00]. This version introduces some new actions while combining the concepts of actions and yielders and completely redesigns the communicative facet. AN-2 is smaller than the original action semantics, however it has not been used as much.

3.10 Conclusion

This chapter introduced the semantic framework of action semantics. The semantic equations from denotational semantics and the entities specific to action semantics of actions, data and yielders have been introduced. Using a simple calculator language as an example action semantics has been shown to be both precise and pragmatic.
Through the years, action semantics itself has grown and changed by incorporating feedback from usages of action semantics. In this work action semantics and the actions, data and yielders as defined and organized in [Wan97] will be used. The approach of action semantics to store and enact action with the mechanism of abstractions will be introduced and used in section 5.3. Section

5.3 also introduces the looping construct from action semantics.
This chapter also concludes the introduction of all the basic concepts that form the basis of using action semantics with MDE. With both MDE and action semantics being explored the two can now be combined.

4 Action Semantics for MDE

4.1 Introduction

The aim of this thesis is to adapt action semantics for model driven engineering. To ease the use of action semantics within model driven engineering, action semantics itself will also be modeled with a MDE approach. This chapter will first describe the design of action semantics for MDE with MDE in section 4.2. Section 4.3 introduces the action semantics metamodel. The tool chain and its tools and artifacts involved with using action semantics for MDE is laid out in section 4.3. Finally some actions in the metamodel that differ from actions as defined by Mosses are explained in section 4.4. The challenges overcome during the adaptation of action semantics from the domain of programming to modeling languages will be addressed in chapter 5.

4.2 Modeling Action Semantics

Action semantics is used to specify the run-time semantics of metamodels. Such semantic descriptions specify the behavior of the execution of models. These semantics descriptions can be used for documentation, reasoning, simulation, verification and validation and code generation. As noted earlier, the usability for designers of semantics require a formalism that can be read and written by the designers and the users of a model as well as a formalism with formal properties to support usage by tools. Modeling action semantics with MDE will also allow the artifacts of action semantics to be used with the Ecore tool chains. These artifacts are both the action semantics descriptions with the semantic equations that define the semantics of a metamodel and the action tree that specifies the execution semantics of a model.

The action semantics metamodel is the metamodel for action semantic descriptions and the compiled action trees. Because of this action semantics metamodel, the action semantic descriptions and action trees are implementation and technology independent artifacts. This thesis provides a compiler and simulator, which uses standard MDE practices. This allows integration with other MDE tools and transformations into code or models.

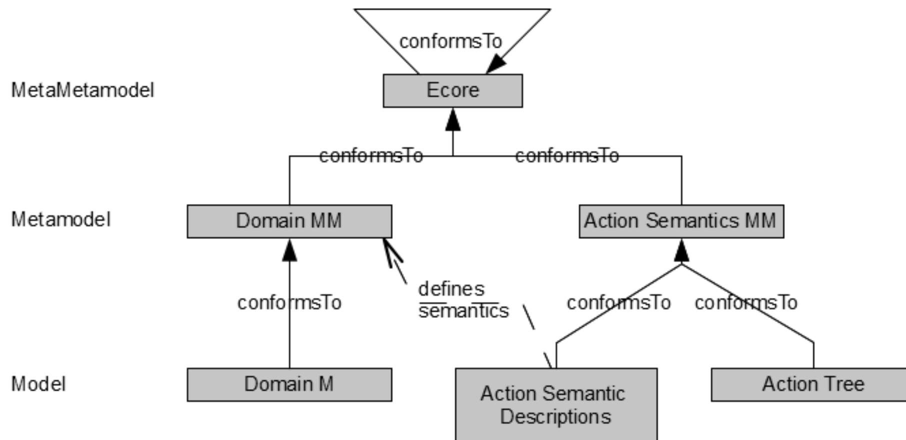


Figure 4.2.1: Action Semantics modeled using Model Driven Engineering

- *Ecore*: The metamodel from EMF, both the domain and action semantics are modeled with this metametamodel
- *Domain modeling language*
  - *Domain Metamodel*: Defines the modeling language to express models in the domain
  - *Domain Model*: A particular model in the domain
- *Action Semantics*
  - *Action Semantics Metamodel*: Defines the modeling language to express action semantics and semantic equations using action semantics
  - *Action Semantic Description*: The specification of the semantics of the domain metamodel consisting of semantic equations that use action semantics
  - *Action Tree*: A tree consisting solely of actions that defines the semantics for a particular model

Fig 4.2.1 gives an overview of how the design of action semantics for MDE mirrors the design of modeling languages with MDE. The conceptual relation defines semantics is between an action semantic description and a metamodel for domain. Through the conforms to relation between the domain metamodel and domain models the domain metamodel and Action Semantic Descriptions together define the semantics for all domain models. Again, we have chosen Ecore as metametamodel for use with the EMF tool chain.

4.3 Action Semantics metamodel

The action semantics metamodel serves as a metamodel for both the action semantic descriptions and action trees. This is because both the action semantic descriptions and action trees require a metamodel for the action, yielder and

data constructs from action semantics. The right hand side of a semantic equation consists of actions and an action trees consists solely of actions. To express semantic equations a couple more constructs are needed to model equations and denotations on the left hand side of an equation.
Therefore, the action semantics metamodel consists of two parts: a part with action notation consisting of actions, yielders, combinators and sorts; and a part to define semantic equations. Fig 4.3.1 shows the action semantics metamodel and these two parts.

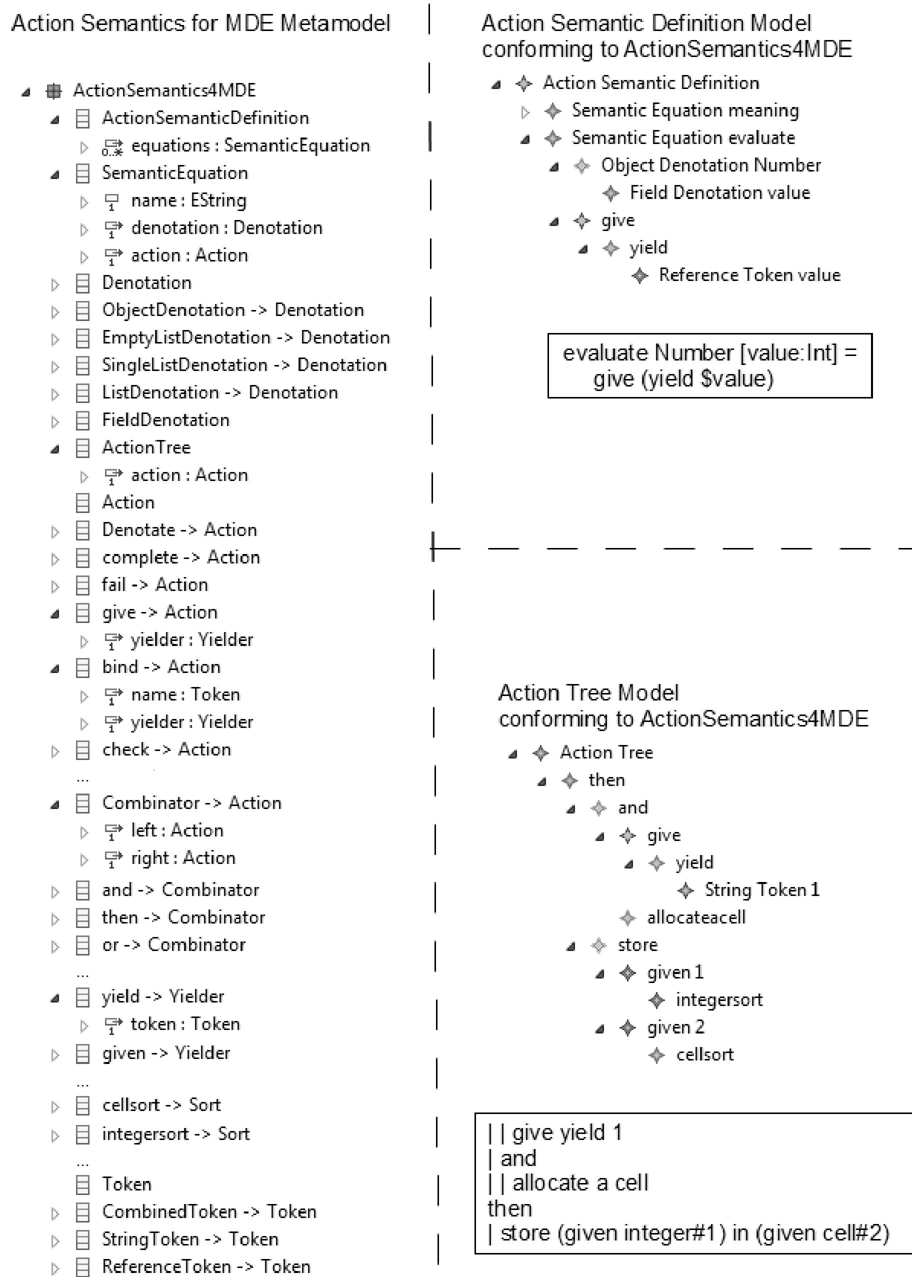


Figure 4.3.1: The Action Semantics for MDE Metamodel with both an Action Semantic Description and Action Tree model conforming to the metamodel

Fig 4.3.1 shows parts from the action semantics metamodel and part of an action semantic description and an action tree. The important parts of the metamodel are highlighted here:
Action semantics metamodel

- *ActionSemanticDefinition*: a collection of semantic equations
  - *SemanticEquation*:
    - *name*: name of the equation, common names are "evaluate" for expressions, "execute" for statements and the top level equation must be called "meaning"
    - *denotation*: class from the domain metamodel the equation is applicable to.
    - *action*: actions on the right hand side of the equation
  - *Denotation*: the name of a class or a type of collection
    - *FieldDenotation*: denotations of attributes or properties of a class that can be referred to on the right hand side of an equation
  - Action and tokens used on the right hand side of a semantic equations:
    - *Denotate*: semantic function call that will return an action, is modeled as an action to allow it to be composed with other actions.
    - *ReferenceToken*: Occurences of a *ReferenceToken* are replaced with the value of the *FieldDenotation* that it refers to. See for example $value in Fig 4.3.1
  - *ActionTree*: A tree of actions that models execution behavior with action semantics
  - All the other construct model the constructs from action semantics as defined by Mosses:
    - *Action*
      - *give*, *complete*, *fail*, *bind*, *etc.*.
      - *Combinator*: combines two actions by directing the flow of information into the sub actions and by merging the results
        and, *then*, *before*, *or*, *otherwise*, *etc.*.
    - *Yielder*
      yield, *given*, *etc.*.
    - *Sorts*
      cellsort, *integersort*, *etc.*.

The two usages for the action semantics metamodel are to model semantic equations and action trees. Therefore the action semantics metamodel consists of a part for semantic equations and actions. The *Denotate* action and *ReferenceToken* that are used on the right hand side of a semantic equation is where these two parts meet and overlap.
On the right hand side of a semantic equation the action notation is mixed with references to the denotation that refers to the domain model. See for example the *$value* token in the semantic equation in Fig 4.3.1. To facilitate this notation

such *ReferenceTokens* are part of action notation in the metamodel. This allows references to the left hand side on the right hand side of the equation where they are mixed or nested within other actions. Because of the way *ReferenceTokens* are modeled in the action semantics metamodel they can also appear in action trees. However conceptually such an appearance would make no sense and it is not allowed, but not enforced by the metamodel. Perhaps a better approach would have been to create a metamodel consisting solely of action notation and a metamodel for denotations that uses the action notation metamodel. This conceptual separation is not enforced as both concepts are in the same action semantics metamodel for practical reasons. Working with one metamodel is easier than working with two metamodels and their composition. Also, as will be explained in section 4.4, the action trees are compiled by a compiler and not hand written, which prevents *ReferenceTokens* to appear in action trees.
The following section explains how the models that conform to the action semantics metamodel are constructed and used. It also explains the tools that are provided to support these usages. Design decisions and challenges that were handled during the construction of the action semantic metamodel and the adaptation from programming languages to MDE are addressed in chapter 5.

4.4 Tools and artifacts

The action semantics metamodel forms the basis of the modeling language that allows the semantics for metamodels to be defined with action semantics. By defining semantic equations for a domain metamodel the semantic specifications for the model in that domain can be constructed. The tools and artifacts that are in this work flow are described here:

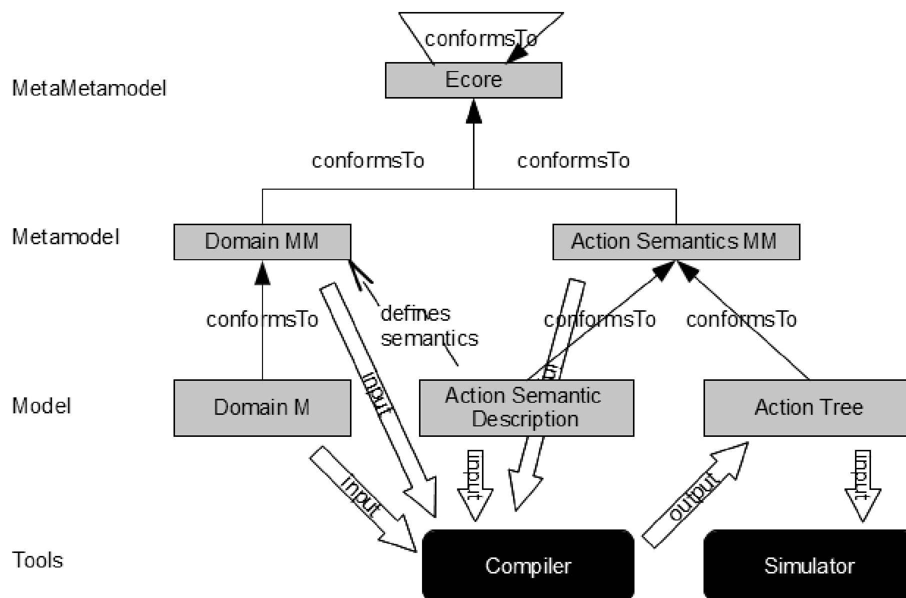


Figure 4.4.1: Tool chain and artifacts

With the artifacts and tools from Fig 4.4.1 the purpose is to allow users of

the domain metamodel and modeling language to construct domain models and to get the semantics for this model in the form of an action tree. Other usages that do not require the tools include using the semantic specification as documentation. The purpose and users of the artifacts and tools from Fig 4.4.1 are as follows:

- Artifacts (Ecore files)
  - *Domain metamodel*: Metamodel that defines a modeling language; created by the designer of this modeling language
  - *Domain model*: A particular model expressed in the domain modeling language; to be created by a user of the modeling language
  - *Action semantics metamodel*: The metamodel for action semantics for MDE; provided
  - *Action semantic descriptions*: A collection of semantic equations that specifies the semantics of the domain modeling language; created by the designer of the modeling language
  - *Action Tree*: A semantic description in the form of a action tree that defines the semantics of the execution of a particular domain model, generated by the compiler
- Tools
  - *Compiler*: Generates an action tree; the compiler is provided
  - *Simulator*: Simulates an action tree, thereby showing the execution semantics of a domain model; the simulator is provided

All the artifacts are Ecore files. These are XMI representations of Ecore models in XML files. Most of these have the ".ecore" file extension, but this is not required. Both the compiler and simulator are tools in the form of jar files, which are provided with this thesis.

4.4.1 Implementation of the Compiler

The compiler applies the semantic equations defined in an action semantic description to an input model conforming to the domain metamodel. The algorithm that the compiler follows is the same approach as in denotational semantics. First the semantic equation with the name "meaning" is applied to the root of the domain model. This requires that every metamodel has a root class and thus every model a root object. It is always possible to introduce such a root class, therefore this is not a restriction on which metamodels can be used with action semantics for MDE. Any application of semantic equations on the right hand side of this equation make this approach construct an action tree. This action tree is written into an Ecore file. The action tree specifies the execution semantics of the domain model. The adaptation of the approach from action semantics (and similar to denotational semantics) for programming languages to action semantics for MDE is further elaborated in chapter 5. The most important adjustment to the algorithm in the compiler is the handling of possible cycles in the graph representation of model, as explained in section 5.3.

The compiler is implemented in Clojure [Clj]. The Eclipse EMF libraries [Eco] are used to work with Ecore files. The inner workings of the compiler is as follows:
Input:

- *Domain metamodel and domain model*
- *Action semantics metamodel and action semantic descriptions, containing the semantic equations*

Both the metamodels are required as input, because the ecore libraries require them to construct a run-time representation of the models. All four files are XML files containing serialized Ecore representations.

- *Step 1*: The semantic equations from the action semantic descriptions are loaded and transformed into a mapping. This mapping consists of a custom representation of the name and type on the left hand side of a semantic equation to the ecore representation of the right hand side.
- *Step 2*: The domain model is loaded and the reference to the root object in the model is found. This is a run-time representation of the ecore file.
- *Step 3*: A *denotator* is constructed. The *denotator* is a function that given the name of a semantic function and an object from a model, will try to find the applicable semantic equation and apply it. The result of such an application is an action tree.
- *Step 4*: The *denotator* is applied given the name of the semantic function "meaning" and the root object in the domain model.
- *Step 5*: When an applicable semantic equation is found for an object the following steps are taken:
  - *Step 5.1*: Any references on the right hand side of the semantic equation are resolved by looking up the reference in the object. This resolves ReferenceTokens, see for instance the $value example in Fig 4.3.1.
  - *Step 5.2*: Any appearance of an application of a semantic equations on the right hand side invokes the *denotator* recursively. This step includes the algorithm to recognize and handle cycles in the model (see section 5.3).
  - *Step 5.3*: The result of applying the *denotator* to an object is an action tree in the form of a run-time representation of Ecore objects, conforming to the action semantics metamodel.

Ouput: The run-time representation of the action tree is serialized into an .ecore file. This is file has the same name as the domain model input file, with "`-tree_gen.actionsemantics4mde`" appended. This file is the semantic specification for the input model.

```
> java -jar compiler.jar
     AS4MDE.ecore CalculatorLang.actionsemantics4mde
     CalculatorLang.ecore model.calculationlang
Wrote Action Tree to:
    model.calculatorlang-tree_gen.actionsemantics4mde
```

Figure 4.4.2: Example execution of the compiler

4.4.2 Implementation of the Simulator

The simulator can visualize the execution of an action tree. The GUI version will show each step in the execution of an action tree by showing the input for each action and the outcome. Another simulator is the command line tool that will not show the intermediate steps but only the result of executing the entire action tree. See Fig 6.2.7 for a screenshot of the GUI and Fig 6.2.8 for an example output of running simulator on the command line. As with the compiler, both versions of the simulator are implemented in Clojure and use the Eclipse EMF libraries. Both tools share the implementation of an action semantics run-time.

Input: *Action tree*
For instance: "`model.calculatorlang-tree_gen.actionsemantics4mde`", which was used as an example for the compiler.

- *Step 1*: The input tree is loaded and each action, yielder and other element in the tree is mapped into a run-time representation. The result of this step is an executable action tree. The run-time representations follow the implementation of action semantics in [Rue93]. Every action is a function from an input state to a new state or failure. Such a state consists of transients, bindings and storage.
- *Step 2*: In order to run the action, the top level action is applied to an empty state, consisting of empty transients, empty bindings and an empty storage. The result of the application of this action is the final state of simulation.
  - *Extra Step for the GUI*: The GUI does an additional step before step 2. It attaches an observer to every action. This allows for the input and output of action to be displayed and for the execution of every action to be paused.

Output: The output is the final state of the top-level action. This is state consisting of transients, bindings and storage, or a *Failure* state.

```
> java -jar simulator.jar
    AS4MDE.ecore CalculatorLang.ecore
    model.calculatorlang-tree_gen.actionsemantics4mde
Transients: [7]
Bindings: [memory --> cell(0)]
Storage: [cell(0) --> 0]
```

Figure 4.4.3: Example execution of the simulator

4.5 Changes to Action Semantics

The action semantics metamodel implements a subset of the actions and yielders as defined by Mosses in [Mos92]. However, the organization into facets from "Modular Monadic Action Semantics" [Wan97] is used. This is done as the work in [Wan97] makes a synthesis of all the various extension and alterations to action semantics that have been introduced after the book by Mosses. "Modular Monadic Action Semantics" includes one action not found in [Mos92] and one action that works differently. These are the *otherwise* and *choose* action. Both are used frequently in the action semantic descriptions that were written for the examples in this thesis, therefore they are explained here.
The *otherwise* action is a combinator action that executes an alternative action in case the result from the other action is failure. With the *otherwise* action the order in which the actions are tried is fixed. This is in contrast to the non-deterministic *or* action combinator that was introduced in the original action semantics. To explain the *otherwise* combinator, first the *or* combinator is introduced:

- A_1 *or* A_2: A non-deterministic combinator. It will non-deterministically choose A_1 or A_2 to execute and will return its result, if the chosen action fails the other action is executed and its result returned
- A_1 *otherwise* A_2: A combinator similar to the *or* combinator, but will always execute A_1 first, and only execute A_2 if A_1 returns failure.

The *choose* action randomly chooses an item from a list. This differs from the *choose* action in the original action semantics where it chooses a random item from a sort.

In "Action Semantics" [Mos92]:

- *choose Sort*: randomly chooses an item from *Sort*

In "Modular Monadic Action Semantics" [Wan97] and the action semantics metamodel:

- *choose Yielder*: randomly chooses an item from the list returned by *Yielder*

4.6 Conclusion

This chapter introduced the action semantics metamodel by modeling action semantics with MDE. The tools and artifacts that are used in the tool chain for this modeling language have been introduced. This tool chain consists of a compiler and simulator and artifacts that need to be written by designers and artifacts that are tool generated. The action semantic descriptions for a domain metamodel need to be written, while the compiler can generate the action tree for a domain model from the model and the action semantic descriptions as input. This action tree is the semantic specification of a domain model. The simulator can be used to run this action tree.
The following chapter will address the challenges that were met while adapting action semantics from programming languages to MDE and the impact this had

on the design of the action semantic metamodel and modeling language and the tools.

5

Challenges in applying Action Semantics to MDE

5.1 Introduction

Action semantics originated as a semantic formalism for programming languages. The aim of this thesis is to apply action semantics to MDE. While trying to use action semantics with modeling languages a couple of challenges appeared due to the differences between the domain of programming languages and MDE. These challenges are:

- Denotations over models instead of grammars and abstract syntax trees (section 5.2)
- Possible cycles in models which are graphs rather than trees (section 5.3)
- Multiplicities (section 5.4)
- Inheritance (section 5.5)

This chapter describes these challenges and their solutions and how these solutions are reflected in the design of the action semantics metamodel and the tools from the previous chapter.

5.2 Denotations over models

Programming languages work with grammars, abstract syntax trees and concrete syntax. Model driven engineering works with metamodels, models and possibly a concrete syntax. Using semantic equations with MDE is explained in this section.

Originally action semantics and the parts of denotational semantics that it uses have been defined for programming languages. Action semantic descriptions for programming languages define semantic equations over the grammar of a programming language. These semantic equations are applied to an abstract syntax tree of a program. Instead of grammars and abstract syntax trees MDE works with metamodels and models, which are graphs. This section explains how semantic equations can be defined over metamodels and be applied to models. This is the first step that was taken during the adaptation of action semantics from programming languages to MDE.

	Programming languages	MDE
Sem. equations defined over	Grammar	Metamodel
Sem. equations applied to	Abstract Syntax Tree	Model (graph)

Table 5.2.1: Semantic equations definition and application domains

This section will introduce semantic equations defined over metamodels that can be applied to models. The signature of these semantic equations will denotate classes in a metamodel and can be applied to elements in a model. Semantic equations are modeled with the action semantics metamodel that was introduced in the previous chapter. The details on how semantic equations are modeled in the metamodel are elaborated in this section. To create a proper understanding of semantic equations an example from programming languages is shown first. Then the semantic equations for MDE are applied to an example modeling language.

Grammar	Abstract Syntax Tree
Statement ::= | "if" Expression "then" Statement | "print" String | Expression ::= | Expression "==" Expression | Expression "+" Expression | Number	if == print + 3 "OK" 1 2 if ((1 + 2) == 3) then print OK

Figure 5.2.1: Simple if statement in a programming language

```
01: execute ["if" E:Expression "then" S1:Statement] =
02: | evaluate E
03: then
04: | | check (the given truth-value 1 is true)
05: | then
06: | | execute S1
07: or
08: | | check (the given truth-value 1 is false)
09: | then
10: | | complete
```

Figure 5.2.2: Semantic equations for an "if" statement in a programming language

The semantic equation in Fig 5.2.3 from [Mos92, p. 28] defines the semantics for a common "if" statement in a programming language. The grammar, an example program and the corresponding AST are shown in Fig 5.2.2. The semantic

equation has on the left hand side a mix of the concrete syntax of an “if” statement as found in the grammar, combined with references to the expression and statement that refer to the nested parts in the AST. These references *E* and *S1* occur again on the right hand side of the equation.

In model driven engineering the metamodel of a language fulfills the function of the grammar of a programming language. A model of a metamodel has the same function as an abstract syntax tree. However both a metamodel and models are graphs and not trees. The grammar of the example programming language in Fig 5.2.2 refers to parts of the concrete syntax of the programming language. The syntax for a modeling language does not occur in a metamodel or model representation.
The concrete syntax of a modeling language, whether it is textual or visual is more distinct part in MDE than it is in programming languages. In fact a modeling language does not necessarily have a concrete syntax. Therefore semantic equations in action semantics for MDE are defined over metamodels and applied to models.

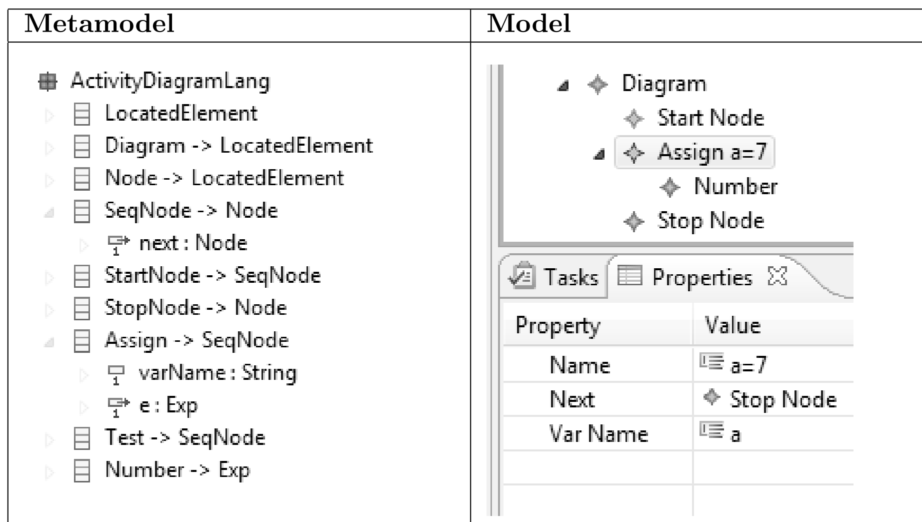


Figure 5.2.3: Metamodel and model for the Activity Diagram language as visualized in Ecore/Eclipse

Fig 5.2.4 shows the metamodel and an example model for the Activity Diagram language that is used as a case study in chapter 6. Again note that there is no concrete syntax for either the metamodel or model, instead the representation as used by the Eclipse editor for Ecore is shown. Alternative representations include showing the XML representation of both artifacts. The following figure demonstrates how semantics equations work with models:

```
01: execute Assign [e:Expr varName:String next:Node] =
02: | | | denotate evaluate $e
03: | | and
04: | | | | give (cell-sort bound to $varName)
05: | | | otherwise
06: | | | | | allocate a cell
07: | | | | then
08: | | | | | | regive
09: | | | | | and
10: | | | | | | bind $varName (given cell-sort 1)
11: | then
12: | | store (given integer-sort 1) (given cell-sort 2)
13: before
14: | denotate execute $next
```

Figure 5.2.4: Semantic equation for an Assign object from the Activity Diagram language

Fig 5.2.5 shows a semantic equation that specifies part of the semantics of the Activity Diagram language by using action semantics for MDE. Instead of using the concrete syntax and structure from the grammar as seen with programming languages in Fig 5.1.3, the semantic equation is defined using the structure and properties of classes in the metamodel. The rest of this section will focus solely on the form of semantic equations in action semantics for MDE and how they are modeled in the action semantics metamodel. The semantics that is specified by the semantic equation for the example modeling language is not important.

A semantic equation has a signature consisting of a name and a class name and the properties of the class that will be used on the right hand side of the equation. The properties for a class can be either attributes or references. The left hand side of the semantic equation in Fig 5.2.5 breaks down into the following parts:

- *evaluate*: The name of the semantic equation
- *Assign*: The name of the class the semantic equation is applicable to
- *[e:Expr varName:String next:Node]*: A collection of properties of a class
  - *e:Expr*: A reference named *e* to an *Expr* class
  - *varName:String*: An attribute named *varName* that is of type *String*
  - *next:Node*: A reference to a *Node* class, this property is defined in the superclass *SeqNode*

In the metamodel for action semantics for MDE (Fig 4.2.2) these parts correspond to *SemanticEquation*, *ObjectDenotation*, and *FieldDenotation*s.

The type and names of the properties used in the semantic equation are the same as found in the metamodel in Fig 5.2.4. In the semantic equation for a programming language in Fig 5.2.3 these names could be freely chosen. Mosses uses capital letters for these names, which make them stand out in the action tree on the right hand side of the equation. The semantic equations for MDE use the names of properties as defined in a metamodel, which may or may not be capitalized. Instead of using capitalization as a cosmetic indicator, the references to properties are prefixed with a $ sign on the right hand side of the equation. The properties of the assign class are both used as arguments to semantic functions calls (*$e* and *$next* in line 02 and 14) and as tokens in actions (*$varName* on both line 04 and 10). The collection of properties of a class does not need to contain all the properties of that class. Only the properties that are used on the right hand side of the semantic equation need to be specified. The name of the class a semantic equation is applicable to is used to see if a semantic equation can be applied to an object.
Also different from the semantic equation for programming languages is the *denotate* keyword that precedes the calls to the semantic equations in line 02 and 14. The semantic equations as introduced by Mosses use no such keyword. It is in the DSL for action semantics for MDE as it makes the parsing of semantic equations vastly easier. The *denotate* keyword and the $ token are both a relic from a prototype of action semantics for MDE that was an embedded DSL. However also in the current tool chain this notation makes parsing easier because the tokens indicate constructs that are not actions and yielders from action semantics but template constructs.
On the right hand side of a semantic equation it is also possible to refer to the object that is matched on by using the *$this* reference. This token is an anaphora and is therefore always available, without needing to specify it with the other properties of a class. The *$this* reference is useful for applying another semantic equation on the same object.

5.3 Cycles

Applying semantic equations to an abstract syntax tree of a program is a straight forward proces, when the semantic equations are well-formed. Doing the same to models in MDE, that are graphs and may contain cycles, might lead to an infinite recursion. This problem is solved by the compiler using the looping constructs from action semantics.

Action semantics gives a finite representation of a program's behavior, even if the program execution might be infinite. For instance, a program using a loop can run indefinitely. However this program conforms to a finite grammar and is represented by a finite abstract syntax tree. The semantic specification for this program is built by using the denotational semantics approach of applying the semantic equations on the abstract syntax tree representation of the program. This results in an action tree that is the semantic specification for the behavior of the program.

With action semantics for MDE, we would like to have the same property of being able to specify infinite behavior with a finite semantic specification. However in MDE the origin of possible infinite run-time behavior can not only arise from explicit looping constructs, but from cycles in the graph of a model as well. For looping constructs in modeling languages that are similar to a common "while" construct in programming languages the existing *unfolding* action from action semantics can be used. For loops that occur due to a cycle in the graph of a model of references and objects the approach from denotational semantics needs to be extended to avoid infinite recursion with the application of semantic functions. This will be done by the compiler of action semantics for MDE, which will use the existing *unfolding* action from action semantics to generate finite descriptions of infinite execution semantics.

5.3.1 Unfolding and unfold

First the *unfolding* and *unfold* actions from action semantics will be introduced, which allow finite expression of possibly infinite actions. This example semantic equation specifies the semantics for a "while" looping construct common in many imperative programming languages:

```
01: evaluate While [guard:Expr body:Statement*] =
02: unfolding
03: | | denotate evaluate $guard
04: | then
05: | | | | check (given truth-sort#1 is true)
06: | | | then
07: | | | | | denotate execute $body
08: | | | | then
09: | | | | | unfold
10: | | or
11: | | | | check (given truth-sort#1 is false)
12: | | | then
13: | | | | complete
```

Figure 5.3.1: Action Semantic Description for the "while" construct

The semantic equation in Fig 5.3.1 follows the obvious pattern of checking the boolean *guard* expression, then executing the true branch which executes the *body* and repeats or executing the false branch which simply *complete*s in this case. The repeated check of the *guard* expression and executing the true branch are accomplished by the *unfolding* action in line 02 that wraps the whole action and the *unfold* action in line 09.

In action semantics, these actions are defined as: "*unfolding A*" performs *A* but whenever it reaches the dummy action *unfold* it performs *A* instead. In semantic descriptions for iterative control flow constructs usually only one *unfold* is nested within an *unfolding* action but the number of occurrences of *unfold* is unlimited. Modeling languages that contain looping construct similar to a "while" loop can use these existing *unfolding* constructs from action semantics for programming languages in the semantic equations.

5.3.2 Problems with cycles in models

In MDE metamodels and models can have cycles from an object to itself through references and other objects. Such cycles are not local to one object and can therefore not be addressed within one semantic equation. The following model is an example of such a cycle, which models a loop in the execution behavior:

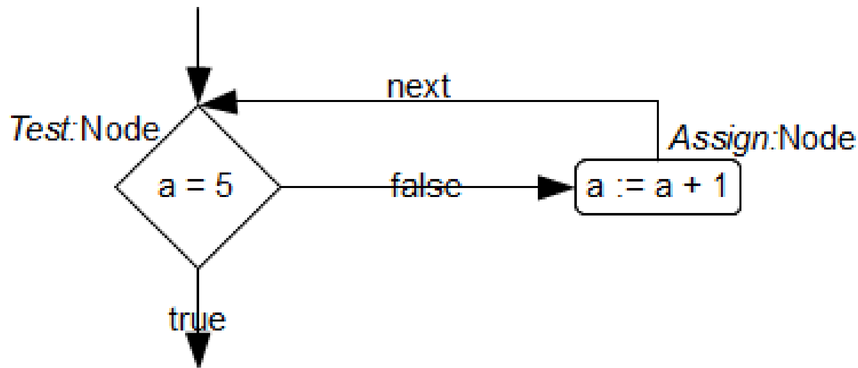


Figure 5.3.2: Visual representation of a cycle in an Activity Diagram

Fig 5.3.2 shows a part of a model of an Activity Diagram, which is part of a case study in chapter 6. The interesting part is the cycle from the *Test* object through the *false* reference, *Assign* object and *next* reference. This part of the graph has similar looping behavior as a common "while" construct, but its behavior is the result of multiple objects and references combined. A model with a *Test* object that does not create a loop would for instance also be possible. This is why specifying the semantics of the looping behavior with the *unfolding* actions in the semantic equation for a *Test* class is not possible.
The compiler for action semantics for MDE takes a model and the action semantic descriptions with the semantic equations as input. The compiler applies the semantic equations on the input model and will recognize when it does so in a loop and will insert the *unfolding* and *unfold* actions to specify the semantics of this loop.

<table>
<tr><th>Semantic Equation</th><th>Action tree</th></tr>
<tr><td><pre>01: execute Test [e:Expr
                  false:Node
                  true:Node] =
02:| denotate evaluate $e
03:then
04:||| check (given truth-sort#1)
          is (yield false)
05:|| then
06:||| denotate execute $false
07:| or
08:||| check (given truth-sort#1)
          is (yield true)
09:|| then
10:||| denotate execute $true</pre></td><td><pre>00: ...
00: ... some actions ...
01:unfolding
02:|| ... some actions ...
03:| then
04:|||| check (given truth-sort#1)
          is (yield false)
05:||| then
06:|||| ... unfold
07:|| or
08:|||| check (given truth-sort#1)
          is (yield true)
09:||| then
10:|||| ... some actions ...</pre></td></tr>
</table>

Figure 5.3.3: Semantic equation that is part of a loop and the resulting Action Tree with inserted $unfolding$ and $unfold$ actions

The template on the right hand side of the semantic equations in Fig 5.3.3 appears in the action tree in the same figure. This right hand side appearing in the action tree is the result of the application of the semantic equation on a *Test* object. The semantic equation *execute* for the *Test* class in Fig 5.3.3 has an application of a semantic equation to the *false* reference in line 06. This semantic equation will at some point apply the semantic equation *execute* to the *Test* object again. This will lead to an infinite recursion. The compiler recognizes this and inserts the *unfolding* and *unfold* actions, as can be seen in the action tree in Fig 5.3.3. The template from the right hand side of the *execute* semantic equation occurs again, but is wrapped by the *unfolding* action in line 01. Somewhere down in the sub tree of this wrapped action the application of the semantic equation *execute* to the *Test* object is replaced by the *unfold* action, as can be seen in line 06. This action tree has the same structure as the “while” construct from Fig 5.3.1, where it was explicitly constructed in a semantic equation.

By having the compiler insert the *unfolding* actions the semantic functions can still be defined over single classes in a metamodel and not be concerned with any nesting they may end up in. This solves the problem of possible cycles in the graph structure of a model that model looping behavior, while defining the semantics for a language can still be done at the level of individual classes.

5.3.3 Compiler algorithm

The algorithm in the compiler that recognizes loops and inserts the *unfolding* and *unfold* actions is as follows. The compiler follows the approach of generating the semantic specification for a program by applying semantic equations to an abstract syntax tree. This approach in action semantics is similar to the approach of denotational semantics. The recognition of cycles and the insertion of *unfolding* and *unfold* actions is combined within this approach. For MDE the compiler applies the semantic equations to a model represented as a graph.

The process is started by applying a semantic equation named “meaning” to the root of a model. Whenever the right hand side of a semantic equation has an application of a semantic equation to a reference of the object, the result of this application is used to construct the action tree. The result of this process is an action tree that specifies the behavior and execution semantics of the input model. The recognition of cycles and inserting *unfolding* and *unfold* actions by the compiler is shown in Fig 5.3.4:

```
    // set of (semantic-equation, domain-object) pairs applied
    // so far in this branch
01: current-denotations := empty
    // set of (semantic-equation, domain-object) pairs whose
    // result will need to be wrapped in an "unfolding" action
02: actions-to-wrap := empty
03: function denotate(semantic-equation, domain-object):action {
04:   if ( (semantic-equation, domain-object)
                          element-of current-denotations)
05:   then {
       // Arrived at the beginning of a loop
       // Insert an unfold action instead and mark the
       // whole application of the semantic equation to
       // the domain object to be wrapped in an
       // "unfolding" action.
06:    actions-to-wrap add (semantic-equation, domain-object)
07:    return unfold;
08:   } else {
       // Store that we are constructing this action
09:    current-denotations put (semantic-equation, domain-object)
10:    action := build-action(semantic-equation, domain-object)
             // build-action fills the action template on the
             // right hand side of the semantic equation with
             // the attributes and references from
             // domain-object, possibly calling this
             // denotate function.
11:    if ( (semantic-function, domain-object)
                          element-of actions-to-wrap)
12:    then {
13:          result := unfolding action
14:    } else {
15:          result := action
16:    }
17:    actions-to-wrap remove (semantic-equation, domain-object)
18:    current-denotations remove (semantic-equation, domain-object)
19:    return result;
20: }
```

Figure 5.3.4: Pseudo code description of the compiler loop recognition and *unfolding* insertion

The compiler uses two functions to construct an action tree from an input model

and semantic equations. The *denotate* function applies a semantic equations to an object in the input model and returns an action, which often consists of nested actions and is then an action tree. The *build-action* function in line 10 fills in the right hand side of semantic equation by resolving any references to attributes or references in an objects. This *build-action* function will also possibly apply semantic equations to references in an object. The rest of the code does book keeping to recognize when such an application of a semantic function results in a loop and returns the appropriate *unfold* action, as seen in line 07. Whenever an *unfold* action gets inserted the whole right hand side of the semantic equation it is nested in gets wrapped by an *unfolding* action, as seen in line 13.

The graph structure of models does not only allow for cycles, it also allows for nested cycles or intertwined cycles to occur in models. Also cycles longer than the loop in the example with two references and objects are possible. The approach shown for single cycles needs a small extension before being applicable to all the different sorts of cycles that may exist in models.

5.3.4 Extension to the unfolding and unfold actions

The result of the compiler algorithm applied to a model with a cycle nested within another cycle will create a nesting of an action wrapped in an *unfolding* action, containing another action wrapped with *unfolding* that contains two *unfold* actions. Each of these *unfold*s corresponds to a different *unfolding*. The semantics for an *unfold* action is to substitute it with the action contained in the surrounding *unfolding* action. For both *unfold* actions this will be the innermost *unfolding* action in this example. This is not the intended behavior for one of these *unfold* action.
A small extension is required to be able to make each *unfold* action correspond to the correct *unfolding* action in case of nested loops. The solution in our approach is to extend the *unfolding* and *unfold* actions with a label to be able to differentiate between nested *unfolding*s. With a label the signatures for both actions become:

- *unfolding label Action*
  performs *Action*, but whenever it reaches the dummy action *unfold label*, it performs *Action* instead
- *unfold label*
  this action will be replaced by the action wrapped by the enclosing *unfolding* action with the same *label*

Semantic equations that use the *unfolding* and *unfold* constructs will need to define a label. The *unfolding* and *unfold* actions that are inserted by the compiler will have a label that is generated by the compiler. *Unfolding* actions are used to prevent infinite recursion from the recursive application of a semantic equation to an object. Therefore the label for this *unfolding* action is generated from these two artifacts to ensure uniqueness of the label for nested *unfolding*s. For a semantic equation a unique identifier is the combination of its name and the name of the class the equation can be applied on. A model has no unique identifier other than its hashcode, unless explicitly modeled in its metamodel.

For that reason the hashcode of an object is used as its identifier for this part of the label.
The label that is generated by the compiler therefore consists of a name and type from the semantic equation and a number from the hashcode of the object. This makes the label not too obscured which is nice as these labels show up in the generated action tree.

This extension stays close to the original *unfolding* and *unfold* action in action semantics, as is evident from the way the *unfolding* and *unfold* actions are implemented in the simulator developed for this thesis. The implementation for both operators is based on the implementation of action semantics in ML for semantic prototypes [Rue93]. This implementation expresses both the *unfolding* and *unfold* actions by only using other actions and yielders from action semantics. Because the implementation is defined by actions and yielders that are already in action semantics, the semantic specification of action semantics itself does not need to be extended to include the extended *unfolding* and *unfold* actions.

```
01: unfolding label Action =
02: | furthermore
03: | | bind label to (abstraction-of Action)
04: hence
05: | Action

06: unfold label =
07:   enact (bound abstraction-sort to label)
```

Figure 5.3.5: Implementation of the extended $unfolding$ and $unfold$ construct with action semantics

The *abstraction-of* yielder and *enact* action in Fig 5.3.5 are part of the reflective facet of action semantics. *abstraction-of* turns actions into abstractions that can be stored in the bindings or storage and retrieved and enacted at run-time. In the implementation in [Rue93] the label in lines 03 and 07 was a constant. In our approach this label is specified explicitly in the *unfolding* and *unfold* action.

With the extension of adding a label to the *unfolding* and *unfold* actions and the compiler algorithm to insert them where needed, it is possible to make a finite semantic description of infinite behavior.

5.4 Multiplicities

Grammars for programming languages allow for options and repetitions. In metamodels these are modeled with multiplicities.

When a grammar of a programming language has an option or repetition in the production rules the nodes in the abstract syntax tree of a program can have a variable number of children. For example, a rule in a grammar for the body of a function can contain any number of statements. In modeling languages

such constructs are modeled with multiplicities of references in the metamodel. Semantic equations match on parts of models conforming to a metamodel. So far all the example semantic equations match only on single objects. The approach to handling multiplicities in references is explained in this section by using the calculator language as an example. The modeling of semantic equations with multiplicities in the action semantics metamodel is also explained.

```
Calculation ::= Expr+
```

Figure 5.4.1: Production rule in the grammar for the calculator language containing a repetition

Figure 5.4.2: Part of the metamodel for the calculator language

The calculator language from chapter 4 had the production rule from Fig 5.4.1 in its grammar, that states that a *Calculation* consists of one or more *Expr*essions. The same part of the calculator language defined by a metamodel is shown in Fig 5.4.2 where the same structure is expressed using multiplicities. The *Calculation* rule in the grammar allows for abstract syntax trees where a *Calculation* node has at least one branch with an *Expr* child. A model conforming to the calculator metamodel will have a *Calculation* object with an attribute called *exprs* containing a collection of one or more *Expr*essions. This structure is equivalent to the abstract syntax tree representation.

In [Mos92, Sec. 2.2] semantic equations for grammars containing optional and repeating clauses are described. The equivalent multiplicities in metamodels are shown in Table 5.4.3:

Grammar rule	**Metamodel reference multiplicity**
? Optional	0..1
* Repeatable	0..*
+ Obligatory repeatable	1..*
	n..m where $0 \leq n < m \leq *$

Figure 5.4.3: Comparison between multiplicities in grammars and metamodels

As shown in the last row of table 5.4.3 the multiplicities for a reference can also be a range between zero and unlimited. Action semantics for programming languages allows for semantic equations that match on multiplicities, similar to denotational semantics. These semantic equations are structured to combine the semantic specification of the items contained in the multiplicities. For an

abstract syntax tree containing a node with a variable number of child nodes this means that the semantic equations handling the multiplicity simply combine the semantic specifications from applying the appropriate semantic equation to each node individually. With action semantics the semantic equations for multiplicities usually consists of a combinator action and semantic equations applied to the child nodes. The notation in [Mos92] uses tuples in the semantic equations that are applicable to multiplicities. Such a tuple consists of a pair of the first child node and the rest of the child nodes. Action semantics for MDE uses the same approach by abstracting collections into having a *first* and *rest* element, similar to the head and tail of a list. This approach is often found in functional and logic programming languages.

Model	**Semantic equation**

Calculation
  Plus
    Number 3
    Number 4
	`01: evaluate [Expr] [first:Expr] =`
	`02:   denotate evaluate $first`

Figure 5.4.4: Semantic equation applicable to a multiplicity of 1

Fig 5.4.4 shows a semantic equation applicable to a collection of size 1 and an example model of the calculator language it is applicable to. The brackets surrounding *Expr* is the syntax for semantic equations that work on collections. This semantic equation “evaluate” simply states that the semantics for a collection of one item is the application of the semantic equation “evaluate” to the first (and only) item of the collection. In the metamodel for action semantics for MDE this denotation over a collection is called a *SingleListDenotation*. The field denotation *first* is the only denotation that is allowed in this kind of semantic equations.

Model	**Semantic equation**

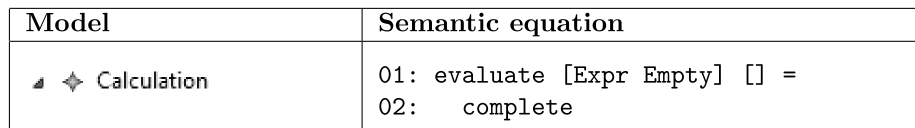
Calculation
  Number 8
  Plus
    MPut
      Number 10
    Plus
      Number 6
      MRec
  Number 12
	`01: evaluate [Expr | Rest] [first:Expr,`
	`                              rest:Expr*] =`
	`02: | denotate evaluate $first`
	`03: and`
	`04: | denotate evaluate $rest`

Figure 5.4.5: Semantic equation applicable to a multiplicity of 2 or more

Fig 5.4.5 shows a semantic equation applicable to a collection consisting of *at least* 2 items and a model conforming to the calculator language it is applicable to. The model is a *Calculation* consisting of 3 *Expr*essions. The three expressions in the calculation are the result of pressing the “=” button after each expression

on a calculator. The semantic equation only consists of the *and* combinator that combines the applications of semantic equations to the items in the collection. The syntax for a semantic equation applicable to a multiplicity of two or more are the brackets surrounding the type name *Expr*, the vertical bar and "Rest". In the action semantics for MDE metamodel this is called a *ListDenotation*. The only allowed field denotations are *first*, which is the first item in the collection, and *rest* which refers the whole tail of the collection. The field denotation *rest* is a collection itself, as indicated by the * following the *Expr* type. The semantic equation for a collection consisting of only one item and a semantic equation applicable to collections of two items or more are sufficient to denotate any collection of any size with at least one item. This uses the destructuring principle for lists that is common in most functional programming languages. The semantic equation `evaluate [Expr | Rest]` will be applied until rest contains just one item, to which the semantic equation `evaluate [Expr]` will be applied.

One final case needs to be addressed for collections that are empty. The calculator language requires a *Calculation* to consist of at least one item. But references that have a multiplicity of zero or more can also occur in metamodels. For example consider the case where the multiplicity of a reference between a *Calculation* and *Expr* is 0..*.



Model	Semantic equation
Calculation	01: evaluate [Expr Empty] [] = 02: complete

Figure 5.4.6: Semantic equation applicable to a multiplicity of 0

The semantic equation in Fig 5.4.6 shows the syntax for matching on an empty collection. In the action semantics for MDE metamodel this is called an *EmptyListDenotation*. The semantic equation here shows that the semantics for a *Calculation* consisting of zero *Expr*essions is the simple action *complete*. The action *complete* returns empty transients and empty bindings and does nothing with any information flowing into the action. There is an explicit notation for empty collections because action semantic descriptions are constructed by combining the application of semantic functions on parts of a model. This allows the application of a function to a collection to always return an action that can be composed with the other parts, regardless of the number of items in a collection.

Our approach has semantic equations for empty collections, collections with one item and collections of 2 items or more. Theoretically an approach with only semantic equations for empty and non-empty collections would suffice. Our approach is more practical because a semantic equation for an empty collection is only required when the multiplicities for the reference includes zero.
The semantic equations for collections are applicable to any collection in an Ecore model. Collections can be references with multiplicities as well as an attribute of a class that is a collection. The semantic equations use the same approach for both because in Ecore both are uniformly modeled with the same *List* interface.

In this whole section collections were considered consisting of generalized *Expr*essions, without needing knowledge of what kind of *Expr*ession it actually is. In the calculator language subclasses of the expressions are for instance the *Plus* and *Number* classes. This structure is very common in object-oriented structures as found in metamodels in MDE. Action semantics and the semantic equations for collection works very well with these structures, because of the approach of action semantics of composing actions with combinators. This allows the composition of the semantics of a collection to be done by combining the semantic specification of the items in the collections, while the semantic equations for collections and classes stay separate.

5.5 Inheritance

Ecore allows for inheritance and generalization following object-oriented modeling principles that imply shared semantics for classes.

Models in MDE are usually designed and constructed using object-oriented analysis and design. Any class or super class encapsulates shared behavior and a class inheriting from such a super class designates that it shares the same features. It would therefore make sense to be able to express the semantics of a model by using denotations over classes at whatever level they are defined.
Currently only denotations in semantic equations over concrete classes are supported in action semantics for MDE. This means that the concrete type of an object is used to find a semantic equation that can be applied to it. The left hand side of a semantic equation must specify the concrete class it is applicable to. Consequently the concrete class of an object is used to find which semantic equations is applicable to it.
An example of a case where a super class defines most of the semantics for its sub classes can be found in the calculator language:

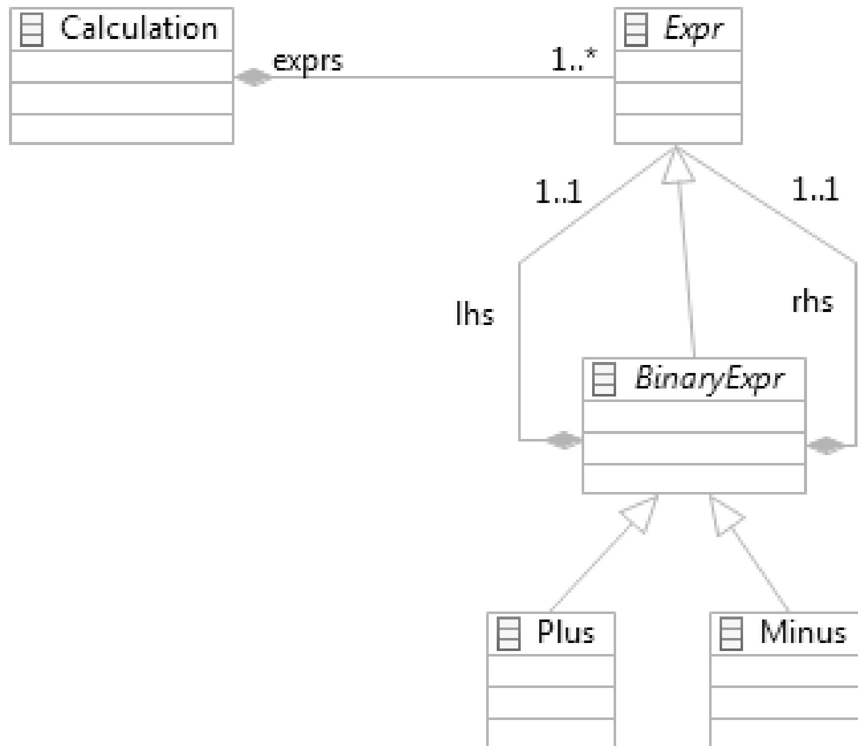


Figure 5.5.1: Part of the calculator language metamodel

The semantic equations for both the *Plus* and *Minus* classes show that the common concepts captured by the *BinaryExpr* class is duplicated:

```
01: evaluate Plus [lhs:Expr rhs:Expr] =
02: | | denotate evaluate $lhs
03: | and
04: | | denotate evaluate $rhs
05: then
06: | give (sum (given integer-sort 1, given integer-sort 2))

07: evaluate Minus [lhs:Expr rhs:Expr] =
08: | | denotate evaluate $lhs
09: | and
10: | | denotate evaluate $rhs
11: then
12: | give (difference (given integer-sort 1, given integer-sort 2))
```

Figure 5.5.2: Semantic equations for expressions in the calculator language

The two semantic equations only differ in the type of yielder that is applied to the values of the left and right hand side of the expression. The actions in line

02 through 05 and 08 through 11 are the semantics for a *BinaryExp*, yet these actions are replicated in both equations. It would be nice if these semantics could be expressed at the class level where they originate. However, in action semantics this is currently not possible. Also note that in line 01 and 07 the semantic equations have the concrete types *Plus* and *Minus* in their signature and the abstract type *Expr*ession in the collection of properties. The applications of semantic equations on the properties *lhs* and *rhs* in lines 02, 04, 08 and 10 will use the concrete class of the properties.

The reason for only allowing semantic equations for concrete types is that the object orientation in Ecore has very few restrictions. A class may for instance inherit from two different classes which inherit from a common super class. This is the so-called “Diamond Problem”. In the Java programming language the problem is prevented by allowing a class only one superclass. If action semantics for MDE would allow semantic equations over non-concrete classes a situation could occur where more than one semantic equation is applicable. To resolve such conflicts a precedence between different semantic equations must be defined or be configurable. The choice to only allow semantic equations for concrete classes is chosen because with it those conflicts can not occur and therefore do not need a construct to resolve them. The down side is that the same semantics need to be repeated, as is the case for *Plus* and *Minus*.

In the previous section semantic equations for collections have been introduced. In Fig 5.4.2 a *Calculation* consisted of multiple *Expr*essions in a collection. This *Expr* class is obviously not a concrete class. However, this class is only used to refer to collections consisting of classes with the *Expr* type. The application of a semantic function to the first item of a collection will still try to find a semantic function that matches the concrete type of that object.

The duplication of semantic specifications for shared behavior is not desired in a design or specification. However, to prevent the conflicts between multiple applicable semantic equations, semantic equations are only allowed on concrete classes.

5.6 Conclusion

This chapter answered the research question “Can action semantics be applied to modeling languages in model driven engineering, with its origin from programming languages?” (R1). The answer to this question with the challenges and solutions outlined in this chapter is yes. The biggest challenge encountered is the possibility of cycles in models addressed in section 5.3. The solution to this challenge was found by having the compiler insert the unfolding and unfold constructs from action semantics into action trees. With the challenges resolved, action semantics can now be used with MDE. The following chapter will introduce two case studies of two modeling languages with their semantics defined with action semantics.

6 Case Studies

6.1 Introduction

The adaption of action semantics for MDE has been guided by a couple of case studies. Most of these examples are the same as used in [Wol09]. All the examples consist of a metamodel, example models and a semantic specification done with action semantics. In this chapter two examples are discussed: a modeling language for simplified activity diagrams and a modeling language for a production line.

6.2 Activity Diagram

The first example is a modeling language for activity diagrams. The activity diagram language is a simple diagrammatic language that can be used to visualize the control flow of programs. It is not the activity diagram language from UML. The example model that uses this metamodel is a simple loop with a counter:

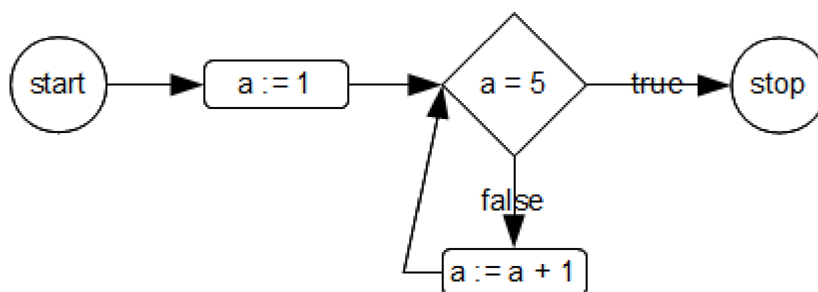


Figure 6.2.1: Visual representation of the count example activity diagram

The count example in Fig 6.2.1 uses a variable “a” that will require the use of the storage and bindings concepts from action semantics. Having these computational concepts available here will be shown to be as useful for modeling languages as they are for programming languages. This small example also contains a cycle that requires the automatic insertion of the *unfolding* construct by the compiler.

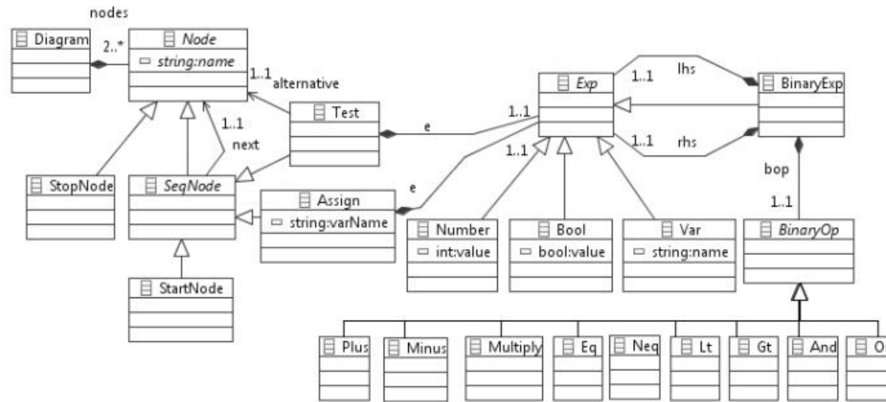


Figure 6.2.2: Activity Diagram Metamodel (from [Wol09])

The metamodel in Fig 6.2.2 has a single root class *Diagram* as required (see section 4.3). Also the possibility of cycles can be seen from the "next" reference from *SeqNode* to *Node*. Most of the right side of the metamodel is concerned with being able to express simple expressions with simple arithmetic and boolean operators.
The semantics for the activity diagram modeling language will be specified using action semantics for MDE. This is done by creating action semantic descriptions consisting of semantic equations. The semantic equations will be used to construct an action tree that specifies the semantics of a model in the activity diagram language by filling in and combining the action templates on the right hand side of the equations. In our framework the semantic equations are expressed over concrete classes or collections in the metamodel of the modeling language (see section 5.2, 5.4 and 5.5).

```
01: meaning Diagram [nodes:Node*] = denotate execute $nodes

02: execute [Node | Rest] [first:Node, rest:Node*] =
      denotate execute $first

03: execute StartNode [next:Node] = denotate execute $next
```

Figure 6.2.3: Semantic equations from the action semantic description for the activity diagram language

The semantic equations all follow the pattern that the semantics of a node consists of the semantics for the node itself and the nodes following it. The semantic equation for all the Nodes will include the application of *denotate execute $next*. Of course the *StopNode* does not include this application. The behavior of an activity diagram starts at the *StartNode*. The first three semantic equations in Fig 6.2.3 expresses this. The mandatory semantic equation named "meaning" is defined for the root object *Diagram*. From the property "nodes" of the *Diagram* node the *StartNode* needs to be found, which is the first *Node* in the collection. This uses the notation for collections and multiplicities in line

02 as introduced in section 5.4. The *$rest* field denotation is not used on the right hand side of the second semantic equation. This is because this equation is only used to apply the *execute* equation to the *StartNode*. All the other nodes in the diagram are reached by following the *$next* reference from each previous node, as seen in line 03.

```
04: execute Assign [next:Node, e:Expr, varName:String] =
05: | | | denotate evaluate $e
06: | | and
07: | | | | give (cell-sort bound to $varName)
08: | | | otherwise
09: | | | | | allocate a cell
10: | | | | then
11: | | | | | | | regive
12: | | | | | and
13: | | | | | | bind $varName (given cell-sort#1)
14: | then
15: | | store (given integer-sort#1) (given cell-sort#2)
16: before
17: | denotate execute $next
```

Figure 6.2.4: Semantic equation for the Assign node

The semantic equation for the *Assign* nodes handles three tasks: the denotation of the following nodes, evaluating the expression on the right hand side of an *Assign* node and storing this value in a variable. The first task is achieved by the "*denotate execute $next*" at the end of the action in line 17. The "next" field of type *Node* is a property from the super class *SeqNode*. As mentioned in the discussion of inheritance in section 5.5 semantic equations are only allowed on concrete types, therefore the "*denotate execute $next*" call will occur in all semantic equations for subtype of the *SeqNode* class. In line 05 the *Expr*ession is evaluated and the result from "*denotate evaluate $e*" will be an integer in the transients and no bindings. Line 07 through 15 store this value from the transients in a variable. The *store* action in line 15 takes care of storing the value for the variable in the appropriate cell. The *given* yielders in the *store* action indicate that the transients passed to this action needs to consists of an integer and a cell.

The evaluation of this integer value and obtaining the cell are independent actions that are combined by the *and* action combinator in line 06. As said, the result from "*denotate evaluate $e*" will be an integer in the transients and no bindings. The result of the other part of the *and* combinator will be a cell in the transients and possibly a new binding for the variable in the bindings. The *and* combinator takes care of concatenating the transients and merging the bindings.

The *otherwise* sub-action involving line 07 through 13 is a bit more elaborate because the semantics for an *Assign* node concerns both the creation and usage of a variable. Whenever an *Assign* node is encountered in an activity diagram the variable that is being assigned to may already exist from an earlier assignment or needs to be created. The *otherwise* combinator will try the left hand side action of *give* first, which will succeed when there is a binding with "var-

Name" in the bindings or will fail when there isn't such a binding. Only when the *give* action returns a failure the second action is executed, which in this case allocates a cell and creates the appropriate binding to the cell. This pattern required the *otherwise* action and is why it is included in the changes to action semantics as explained in section 4.4.

The semantic equation for *Assign* is a good example of how the runtime of action semantics is modeled and used and how combinators can be used to keep sub-parts of actions specific and contained.

```
18: execute Test [e:Expr, next:Node, alternative:Node] =
19: | denotate evaluate $e
20: then
21: | | | check (given truth-sort#1) is (yield true)
22: | | then
23: | | | denotate execute $next
24: | or
25: | | | check (given truth-sort#1) is (yield false)
26: | | then
27: | | | denotate execute $alternative
```

Figure 6.2.5: Semantic equation for the Test node

The *Test* node is the most interesting in an activity diagram as it has two successor nodes. Similar to the semantic equation of the *Assign* node first an action to evaluate the expression is denotated in line 19. This denotation will produce a boolean value in the transients. The *check* actions in line 21 and 25 will complete if the *is* yielder is true and will fail when it is false. This *fail* construct is used to guide the choice of the non-deterministic *or* combinator. The *or* action will try one sub-action and returns the result of this sub-action when it completes. If the sub-action fails the other sub-action is tried. The order in which the sub-action are tried is non-deterministic, as specified in the original action semantics. In the *Assign* node we used the *otherwise* combinator that does guarantee the order in which the sub-action are tried. This *otherwise* combinator was introduced in improvements of action semantics [Wan09] (see also section 4.4).
A *Test* node in an Activity Diagram can be the source of a possible cycle, that has looping behavior. However in this semantic equation we do not have to take care of such possibility. This will be done by the compiler, as explained in section 5.3.

```
28: execute StopNode [] = complete

29: evaluate Var [name:String] =
      give (integer-sort stored in (cell-sort bound to $name))

30: evaluate Number [value:Int] = give (yield $value)

31: evaluate BinaryExp [lhs:Exp, bop:BinaryOp, rhs:Exp] =
32: | | denotate evaluate $lhs
33: | and
34: | | denotate evaluate $rhs
35: then
36: | denotate evaluate $bop

37: evaluate Eq [] =
      give (given integer-sort#1) is (given integer-sort#2)

38: evaluate Plus [] =
      give (sum (given integer-sort#1)  (given integer-sort#2))
```

Figure 6.2.6: Semantic equations for the other nodes in the count example activity diagram

The last few semantic equations in Fig 6.2.6 are mostly denotations for expressions. In the action semantic descriptions these semantic functions are all named "evaluate". The semantic functions that denotate nodes in a diagram all have the name "execute". These names for denotations over expressions and statements is common within action semantic descriptions. The semantic equation for a *StopNode* in line 28 only inserts the *complete* action and does not contain any other denotations. At this node the application of semantic functions will stop. Fig 6.2.3 through 6.2.6 form the complete action semantic description for the activity diagram modeling language, although some semantic equations for expressions that are not used in the count example are omitted. Together with the loop insertion by the compiler this complete specification is enough to built semantic specifications for models of the activity diagram modeling language.

With the action semantic description for the activity diagram modeling language the semantic specification for the count example can be constructed. The following four artifacts are needed as input for the compiler:

- *Action Semantics Metamodel*: Fig 4.2.2
  This metamodel is a static artifact as explained in section 4.3 and can therefore technically be embedded in both the compiler and simulator, however, for practical and illustrative purposes it is explicitly needed and mentioned as input.
- *Action Semantic Description*
  The semantic equations for the activity diagram modeling language are in Fig 6.2.3 through 6.2.6.
- *Activity Diagram Metamodel* Fig 6.2.2

- *Activity Diagram Model*
  For this case study it is the count example from Fig 6.2.1

The compiler will apply the semantic equations in the action semantic descriptions to the count example model. Together with the algorithm for cycles in models from section 5.3 this will produce an action tree that conforms to the action semantics metamodel. This action tree is the semantics for the count example. The whole action tree is shown as visualized by the simulator:

Figure 6.2.7: AS4MDE Simulator Showing the action tree example for the count example

Fig 6.2.7 shows the semantics of the count example with the action tree as shown in the AS4MDE Simulator. This simulator is provided with this thesis and can visualize the execution of actions. The screen shot of the simulator shows the action specification being run. The highlighted *unfold* action in line 42

designates that this action is about to be executed. Before an action is executed it is shown in yellow-orange in the simulator, after an action completes it is shown in green, when it fails it is shown in red.
The *unfolding* and *unfold* actions in line 13 and 42 are inserted by the compiler as explained in section 5.3.
The "before" and "after" panels show the input and result state for an action. In this case there is no output state, as the action hasn't run yet. The "before" state contains the following:

- *Transients*:
  Empty transients; there are no intermediate values passed to this action
- *Bindings*:
  "UF_executeTest13610289", this token maps to an abstraction. It holds the abstraction of the action wrapped by the *unfolding* action in line 13. The "UF_" preamble is inserted by the compiler to mark it as generated. The name "executeTest13610289" is the semantic equations name and the type and hashcode of the object in the model that caused a loop. "a", this token maps to a cell that is in the storage. "a" is the *$varName* and *$name* property used in the semantic equations for *Assign*, *Test* and *Var* in Fig 6.2.4, 6.2.5 and 6.2.6
- *Storage*:
  The storage contains one cell, currently holding the value 2. This means that the executing of this semantic specification initialized "a" to 1 and that the *Assign* node "a = a + 1" has been visited once, before arriving at the *unfold* action to model the loop.

The abstraction bound to the "UF_executeTest13610289" is in the bindings, while the variable "a" has an indirection to the cell in the storage. This is because the abstraction will not change once put in the bindings, while the binding for "a" will remain the same, but the value it maps to can be changed. Running the whole specification will result in this final state:

```
Transients: []
Bindings: [a --> cell(0)]
Storage: [cell(0) --> 5]
```

Figure 6.2.8: Outcome of running the whole semantic specification for the count example

First, it should be noted that this is indeed the desired outcome as it shows the variable "a" to be 5. Second, the binding for the unfolding abstraction is no longer in the bindings. This is because this binding only exists within the scope of the nesting of the *unfolding* action. The final result for the whole semantic specification is the state that is the result of the top level *before* action in line 12.

The action tree for the semantic specification contains a part that will never be executed during the simulation. These are the actions in line 34 through 38. These actions are in the action tree as a result of the application of the semantic equation execute to the *Assign* node "a := a + 1". Because there are two *Assign*

nodes in the count example the template from the semantic equation "execute Assign" from Fig 7.2.4 appears twice in the action tree in line 01 through 13 for a := 1 and line 31 through 41 for "a := a + 1". As said earlier the semantic equation "*execute Assign*" needs to consider both the case that the variable is already initialized and the case that the variable does not exist yet in the bindings and storage. It is not uncommon for such unreachable actions to occur in action trees. This is because semantic equations only consider the object they are applied to and not the composition of this object with other objects in a particular model. There may be models that cause the execution of both cases. For instance, when a variable is reinitialized within a loop. The simulator also shows in the top right corner the number of steps that have been taken to arrive at this action. This number is not fixed when there is a non-deterministic action such as *or* in a semantic specification. During one run it might always first check a branch that will fail whilst other times always choosing the branch that will succeed.

Appendix B contains two other example models of the activity diagram language. These examples have nested cycles.

This case study shows how semantic descriptions for a metamodel can be used to construct the semantics for a model. It also demonstrated how the semantic equations can be kept simple and human readable and how the compiler compiles these into an action tree. The semantic equations for each class in the metamodel of the activity diagram language are all self-contained and only use the constructs from action semantics that they require. For instance, the semantic equation that evaluates an expression only uses the transients while the semantic equation for the *Assign* node combines uses of the transients, bindings and storage. This case study did not only show how action semantics can be used with MDE but it also contained the challenge of possible cycles in models. The approach in the compiler outlined in section 5.3 proved to be sufficient for this example.

6.3 Production Line Language

The second case study is the production line language example from [Rom07]. The example is a domain specific modeling language (DSML) using a visual syntax for models. The work in [Rom07] uses the example to illustrate their approach for formal and tool support for MDE with Maude [Cla07]. The example is used here only to illustrate a possible usage of action semantics with MDE and not to compare the two approaches. All the examples shown so far have mostly been programming languages modeled with MDE, with the exception of the possible loops in the activity diagram language. The production line language also models computation but with some abstractions that have a less straight forward mapping onto computations than the examples seen so far. The goal of this case study is to show that action semantic can also be used for specifying the dynamic semantics for modeling languages that are different from common programming languages, for which action semantics was originally created.
The production line language is a modeling language for production lines that create hammers from a head and a handle. The production line consists of ma-

chines that can generate heads and handles and machines that can assemble a head and a handle into a hammer. The production line is not completely automated as there is no automatic transportation of the output from one machine into another. Machines use trays to get their input from and to store their output in. A human operator moves between trays to move parts from one tray into another. Finally, both the machines and trays have a certain production and storage capacity.

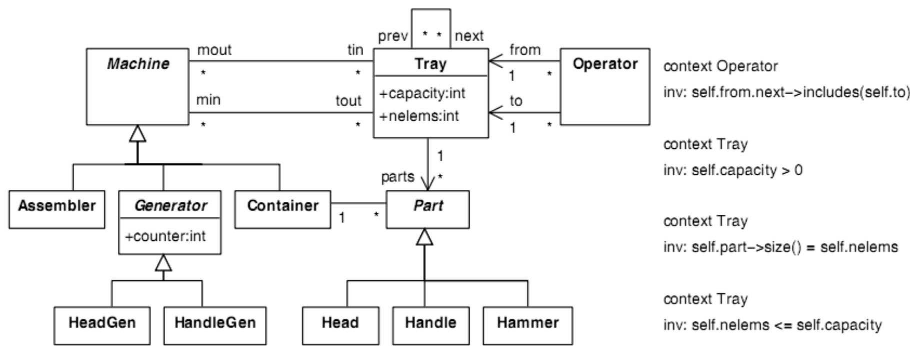


Figure 6.3.1: Production line language metamodel (from [Rom07])

Fig 6.3.1 shows the metamodel for the production line language. The metamodel also contains constraints that are not expressed with the object-oriented structure of the metamodel, but are expressed with invariants on the right hand side of the metamodel. These constraints define that an operator can only move parts between connected trays and they define the capacity limits of trays. The example model of a production line language used for this case study is shown in Fig 6.3.2:

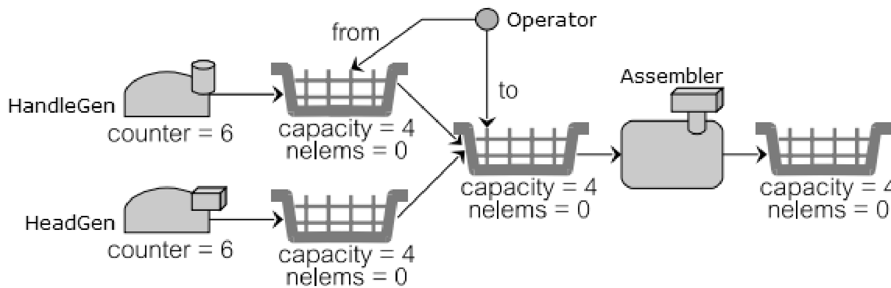


Figure 6.3.2: The hammer production line (from [Rom07])

Fig 6.3.2 shows the model of the hammer production line. On the left side the *HandleGen* and *HeadGen* generators produce *Handle* and *Head* parts in their trays. The *Operator* moves parts into the input tray for the *Assembler*. Finally, the *Assembler* produces *Hammer*s in its output tray. This example model is particularly interesting due to the capacity that is chosen for the generators and trays. The *Assembler* takes a *Head* and *Handle* part from its input tray to produce a *Hammer*. When its input tray is filled to its capacity with either four

*Head*s or four *Handle*s the *Assembler* cannot create any more *Hammer*s. This is a possible deadlock in the production line, because the *Operator* is defined to only move parts into trays following the connecting arrow between the *Tray*s. This possible scenario is the reason this domain specific modeling language was used as an example in [Rom07], which focuses on verification of modeling languages.

The approach we use in defining the semantics for the production line language is inspired by the approach taken in [Rom07]. In [Rom07], the semantics for the production line language is defined using graph transformations. The semantics consists of a number of graph rewriting rules that modify a representation of the model. This is a graph of the model shown in Fig 6.3.2. Every rule has a guard and an action part. When a guard matches on the current graph representation of the model, its action can be executed. Usually this action modifies the graph, thereby enabling other rules. When multiple rules are enabled one rule is randomly chosen to be executed.
The semantic specification of the production line language with action semantics uses the same approach with rules consisting of a guard and action pair. However, action semantics does not have the built-in mechanism of checking and enacting rules, as it is in graph transformations. Therefore, the semantic equations for the production line language constructs a main loop that performs the checking and enacting of rules explicitly. Action semantics has a run-time model consisting of transients, bindings and storage. This run-time does not include a representation of the input model. There is no static or modifiable representation of the input model like there is in the graph transformation approach. Therefore, the parts from the model that are dynamic are explicitly modeled in the semantic equations by creating bindings and storage for them.

The action semantic definitions for the production line language are in appendix [[??? to be included when the as dsl/notation is done]]. The approach for specifying the semantics with action semantics will be illustrated by explaining a run-time state of the action tree for the example hammer production line as visualized in the simulator.

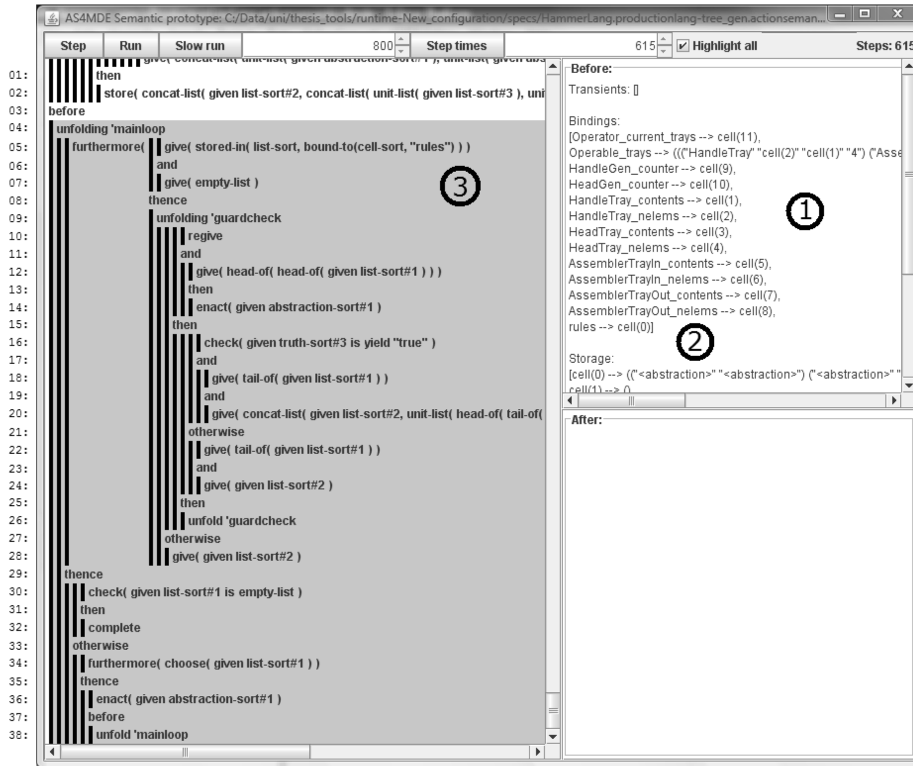


Figure 6.3.3: The hammer example action tree in the simulator

Fig 6.3.3 shows the action tree that has been compiled by applying the semantics equations from the action semantic descriptions for the production line language to the hammer example from Fig 6.3.2. The specification of the semantics for the production line language with action semantics will be illustrated with the three highlighted, numbered parts of Fig 6.3.3.

1. Explicit mapping of dynamic parts of the model into the bindings and storage
2. Rule consisting of a guard and action for every machine and operator in a production line
3. Main loop checking and enacting rules

1. Explicit mapping of dynamic parts of the model into the bindings and storage
Action semantics has explicit constructs for computational concepts such as control flow, bindings and storage. The input model conforming to the metamodel is not part of these concepts and the model cannot be referred to or used neither statically nor dynamically. As can be seen in the workflow in Fig 4.3.2, the compiler takes a model and semantic equations as input and constructs an action tree, which is the semantic specification of the input model. In this compilation step the input model is only statically used. Therefore the semantic equations for machines and trays explicitly put their properties in the binding and storage, the result of which can be seen at highlight 1 in Fig 6.3.3. These

properties named "_counter", "_contents" and "_nelems" model the production capacity of machines and the parts and total number of parts in a tray. This is different from the approach of graph transformations that is used in [Rom07] where the model is available and modifiable during execution. With action semantics, parts of the model are explicitly modeled with bindings and storage. This is the same approach that was used to specify the semantics of a variable in the activity diagram example from section 6.2.

2. Rule consisting of a guard and action for every machine and operator in a production line
In both the informal explanation of the semantics of the production line language and the formal graph transformation approach used in [Rom07] the semantics for a machine and operator are an action that can be performed when the production line is in a certain state. Machines can produce a *Head* or *Handle* when there is room in their output tray to put it in and their own capacity is sufficient. Likewise an operator can move a part from one tray to the next when there is such a part and a tray with sufficient room for the part. The operator can also move to operate other trays when other trays have parts to move. Highlight 2 in Fig 6.3.3 shows these rules in the bindings and storage. The bindings is named "rules" and in the storage it can be seen that the rules consists of a list of pairs of abstractions. These two abstractions are the guard and action for a rule. The semantic equations for machines and operators inserted the actions into the action tree that create and store these abstractions. Thus the semantics for machines and operators are expressed as creating a rule that has a guard and action. The guard of a *HeadGen* machine uses the current state of bindings and storage to check if the corresponding action is applicable. The guard checks if its "HeadGen_counter" capacity is greater than zero and if its output tray has a "HeadGen_nelems" count less than its capacity. The corresponding action actually changes the state by reading and writing to the storage to set the different counters and contents to the appropriate values.
Recall that abstractions are a construct from action semantics that enable actions to be stored in the bindings or storage and they can be retrieved and enacted, as seen earlier in section 5.3.

3. Main loop checking and enacting rules
As said, the checking and enacting of rules is explicitly modeled. The actions that model this are shown at highlight 3 in Fig 6.3.3. This whole highlighted action below the *before* action (line 03) is a loop that checks all the rules that are enabled and puts the corresponding actions in the transients (line 05-28). From these enabled actions in the transients one abstraction is chosen (line 34) and executed (line 36). This main loop is the basis of the whole semantic specification of the production line language. It is therefore part of the top level semantic equation named "meaning" that is applied to the root object in a production line language model.

6.3.1 Possible improvements

This case study shows that action semantics can also be used for a DSML, which has a different model of computation than the straightforward programming languages in earlier examples. The constructs from action semantics as defined by

Mosses and the adaptations for MDE from this work were sufficient to create the semantic specification. This case study also showed possible improvements to action semantics. These improvements are better support for storing parts of a model in the storage, functional programming constructs to replace usages of the unfolding construct and semantic equations for abstract classes.

Improvement 1: Storing parts of a model in the storage
Highlight 1 in Fig 6.3.3 shows the parts of the model that are dynamic need to be explicitly mapped into the bindings and storage. In this case study this is done be creating tokens such as "HeadGen_counter", which is a string combining the name of machine and its property. In the semantic equations this appears a couple of times with the following actions:

```
elaborate HeadGen [id:String, counter:int, min:Tray, mout:Tray] =
| | allocate a cell
| then
| | | store (yield $counter) (given cell-sort#1)
| | and
| | | bind ($id & "_counter") (given cell-sort#1))
...
```

Figure 6.3.4: Common pattern to create a binding for a property of an object

```
(give (stored-in (bound-to cell-sort ($id & "_counter")))
```

Figure 6.3.5: Common pattern to retrieve a binding for a property of an object

Fig 6.3.4 and 6.3.5 show the pattern of actions that is common when encoding a property of an object into the bindings and storage. A cleaner approach would be to introduce new actions and yielders in action semantics that can create these bindings and memory cells in one step. For instance *store-mapping $counter* to replace fig 6.3.4 and *give (stored-mapping $id.counter)* to replace fig 6.3.5. A whole new facet for action semantics that deals with mapping models into the bindings and storage could be introduced.

Improvement 2: Functional programming constructs to replace usages of unfolding
The semantic equations for the production line language use the *unfolding* loop and list data construct from action semantics at various places. Most of these loops are low level implementations of the *map*, *filter* and *reduce* (*foldl*) higher-order functions from functional programming. Because the loops are low level implementations, some of the actions that are used to initialize and break from loops obscure the semantics the actions try to define. The semantics of the higher order functions *map*, *filter* and *reduce* are well-defined and commonly understood, therefore using these constructs instead of explicit loops would result in smaller semantic functions and action trees while defining the same semantics.

Improvement 3: Semantic equations for abstract classes
Finally, the production line language metamodel has a *HeadGen* and a *Handle-*

Gen machine which both inherit from the *Generator* abstract class, as shown in Fig 6.3.1.These machines share the semantics of generating a part of a hammer. This shared semantics is captured in the common superclass *Generator*. In section 5.5 it is explained that semantic equations over abstract classes are not possible. Therefore the semantic equations for the *HeadGen* and *HandleGen* are identical, except in the places where a *Head* or *Handle* occurs. This is another example why the other approaches concerning inheritance mentioned in section 5.5 should be explored.

6.4 Conclusion

The activity diagram and production line language case studies show that action semantics for MDE can be used to define the semantics for both of these modeling languages. The activity diagram case study answers the first research question (RQ1, section 1.3) affirmatively. Action semantics can indeed be adapted for use with modeling languages. The production line language case study is a domain specific modeling language with semantics on a higher level of abstraction than seen in earlier examples. MDE generally uses a higher level of abstraction than programming languages, for which action semantics was originally designed. The successful production line language case study positively answers the second research question (RQ2, section 1.3): Action semantics can be used to specify the semantics of modeling languages that use a higher level of abstraction than programming languages. The two case studies also introduce possibilities for improvements. Most of these improvements are the introductions of new actions and yielders to make the semantic equations more succinct.

7 Conclusion

7.1 Introduction

In this thesis we adapted the semantic framework of action semantics to specify the execution semantics of modeling languages. The result is action semantics for MDE that incorporates the changes and adaptations required to make action semantics work with the higher level of abstraction within MDE in relation to programming languages, where action semantics originated. The thesis provides a compiler and a simulator that can compile and simulate semantic specifications of models. The next section gives a summary of the thesis. Then the results are evaluated and discussed. The final section discusses open issues and future work.

7.2 Summary

Chapter 1 outlines the motivation and background for this work. Model driven engineering (MDE) is the software engineering practice of creating modeling languages to solve problems in a particular domain. The semantics of a language, whether it is a natural language such as English, a programming language or a modeling language, specifies the meaning of a sentence, program or model expressed in that language. Modeling languages require a semantic framework to define their semantics with. The specification of the dynamic semantics for modeling languages is an open issue, for which we propose to adapt action semantics.
Chapter 2 introduces the basic concepts of MDE and semantic formalizations. A modeling language created with MDE is defined by a metamodel and instances of that language are models that conform to the metamodel. Metamodels define the structure of a modeling language in the same way a grammar does for a programming language. Similarly, the notion of an abstract syntax tree in programming languages is a model represented as a graph in MDE. The final sections of chapter 2 introduce semantics. The formal definition of the semantics of a language is required to be able to reason about a language. For this purpose semantic framework needs to be unambiguous, complete and consistent. The semantics for modeling languages created with MDE are used for reasoning and documentation, implementation, code generation and verification and validation.
Chapter 3 introduces action semantics. Action semantics was created by Peter Mosses in '92 as a framework for the formal description of programming lan-

guages. The aim of action semantics is to produce formal semantic specifications that reflect ordinary computational concepts and are easy to read, understand and compose. Thus a formalization framework that is both precise and pragmatic. Action semantics uses the same approach from denotational semantics with semantic equations that map constructs of a language onto its semantic specification. In action semantics this mapping is onto action notation. Action notation has the first class entities actions, data, yielders and a model for the run time of a computation consisting of transients, bindings and storage. Actions and yielders all have a very specific purpose and can be combined with combinators to express more elaborate semantics.
In chapter 4 action semantics itself is modeled with MDE. The action semantics metamodel incorporates the changes to action semantics that are required for the adaptation from programming languages to modeling languages. The metamodel is also the center of the work flow of specifying the semantics for a modeling language. The action semantic description of a language consisting of semantic equations can be fed to the compiler together with a model conforming to the language's metamodel. The compiler will generate an action tree, which is the semantic specification of the semantics of the model. The simulator can be used to visualize the execution of action trees.
The adaptations to action semantics for MDE from its origin with programming languages are explained in chapter 5. A common theme is the usages of models within MDE which are graphs, while programming languages uses grammars and abstract syntax trees. First denotations over models are introduced. The biggest challenge is the possibility of cycles in the graph representation of models. The solution to this challenge is to have the compiler recognize the recursive application of semantic equations and to resolve this by inserting looping actions in the action tree to model these cycles. This approach reuses the existing looping constructs from action semantics. Then how semantic equations for multiplicities work is explained. Finally, the decision to only allow semantic equations over concrete classes is elaborated.
Chapter 6 discusses two case studies of modeling languages with their semantics defined with action semantics for MDE. Models of the activity diagram language can contain cycles and is used to illustrates how the compiler solves the recursive application of semantic equations. The production line language is a domain specific modeling language and has a model of computation that is of a higher level of abstraction than the earlier examples, which were mostly programming languages modeled with MDE. Both case studies show that the adaptation of action semantics from programming languages to MDE as shown in this thesis can be used to define the semantics of modeling languages.

7.3 Evaluation

The aim of this thesis is to explore if action semantics can be applied to MDE to provide formal, understandable, modular and usable semantic specification of models. We will evaluate this work with relation to this aim by revisiting the research questions that guided this effort.

- RQ 1: Can action semantics be applied to modeling languages in model driven engineering, having its origin from programming languages?

  - RQ 1.1: Model driven engineering languages represent models as graphs, programming languages use abstract syntax trees. What adaptations are needed for using graph structures?
  - RQ 1.2: Programming languages use grammars whereas model driven engineering uses metamodels. How can action semantics be used with inheritance, attributes, references and multiplicities as used in metamodels?

In the context of this work RQ 1 is answered affirmatively by the two successful case studies in chapter 6. More specifically, MDE and modeling languages can be seen as a higher level of abstraction in relation to using programming languages to built software systems. Action semantics was developed for use with programming languages and therefore the differences between programming and modeling languages needed to be addressed. Programming languages use grammars and abstract syntax trees to represent programs, while modeling languages use metamodels and models, which are graphs. With modeling languages there is also a more stricter separation between the abstract and concrete syntax of a language. The implications of working with graphs rather than trees is the subject of RQ 1.1 and is answered in section 5.3. The recursive application of semantics equations on trees, which is an approach action semantics shares with denotational semantics, does not necessarily finish on models in modeling language that are graphs. The solution to this problem is to have the compiler insert the looping constructs from action semantics when an infinitely recursive application of a semantic function is found. The compiler is part of the tools and artifacts for action semantics for MDE (section 4.4).
The definition of a modeling language with its metamodel also includes inheritance, attributes, references and multiplicities, which are not used or used differently in the definition of programming languages with grammars and syntax. This is the subject of RQ 1.2 and is answered in sections 5.2, 5.4 and 5.5. Action semantics uses semantic equations to map parts of a language onto its semantic definitions. Semantic equations in action semantics for MDE are defined for classes in a metamodel and this handles the issues of inheritance, attributes, properties and multiplicities.

- RQ 2: Action semantics is a good fit for programming languages because the entities map more closely to computational constructs than the mathematical approaches. Which level of abstraction of computational constructs in action semantics is needed for model driven engineering?

Peter Mosses, the creator of action semantics, introduced constructs to model the common computational concepts of control flow, bindings and storage (see chapter 3). In other semantic frameworks these constructs were often obscured by a mathematical notation. The explicit run time model that includes bindings and storage makes it straight forward, for instance, to define the semantics of creating and using a variable. This makes action semantics a good fit for programming languages where these types of constructs are often found. The abstractions in a modeling languages do not necessarily match the run time model in action semantics. Most of the example languages in this thesis are programming languages modeled with MDE, for which action semantics worked well. The production line language case study from section 6.3 is a modeling

language with semantics on a higher level of abstraction. Also for this modeling language action semantics proved to be expressive enough, although the case study also showed possible improvements.

- RQ 3: What tool support is needed to allow action semantics to be used in model driven engineering?
  - RQ 3.1: How can the behavior or dynamic semantics of a model be shown by using action semantics and a tool?

Action semantics for MDE is itself a modeling language. Therefore it is modeled with MDE and this resulted in the action semantics metamodel (section 4.3). Both the action semantic descriptions for a modeling language consisting of semantic equations and action trees, which are the semantic specification for a model, conform to the action semantics metamodel. Action semantics for MDE is modeled with Ecore and can therefore be used with the Eclipse tool chain. A compiler has been built that applies the semantic equations from an action semantic specification of a modeling language to a model of that language and creates an action tree. The compiler contains the algorithm to solve the possible loops in the graphs representations of models (see section 5.3). The action tree is the semantic specification of the model. The simulator (section 4.4) can be used to run and visualize the action tree. Together the tool support consists of using Ecore and the compiler and simulator. This tool support was sufficient to create prototypes for the two case studies in chapter 6. Action trees are Ecore files and can therefore serve as input to other tools or transformations, for instance code generators.

7.4 Comparison to Related Work

This work consists of using action semantics to specify the semantics for modeling languages. It will be compared here to related work regarding action semantics and other semantic formalisms for specifying semantics for modeling languages.

Action Semantics
Action semantics was created to specify the semantics of programming languages. The same tool support that is desired of semantic frameworks for modeling languages has been realized for programming languages by using action semantics. This work provides a compiler that can compile a semantic specification of a model from the semantic equations, the metamodel and model as input. The result is an action tree that can be simulated in the simulator. Actress [Bro92], Albaco [Mou99], Cantor [Pal92] and OASIS [Orb94] all provide tool support in the form of compiler generation based on action semantic descriptions. They all create a compiler from the semantics and syntactic definition of a language as a stand-alone compiler for that language. The compiler in our work interprets the semantic equations for a modeling language before compiling a model of that language. The related work varies in the amount of analysis that is done to create optimizations in both the compiler for a language and the resulting bytecode or machine code for models. Our work does no optimizations. The simulator from this work serves as an interpreter of action trees

and there is no compiler to compile the action trees into executable code.

Action semantics has first class entities for computational constructs such as control flows, bindings and storage. This level of abstraction makes it easy to create interpreters for the action notation used in action semantics. [Mou93] describes the "Action Notation Interpreter" (ANI) and [Rue93] implements action semantics in Standard ML. The simulator in this work is similar to both. The layout of the simulator with the before and after state of each action and the highlighting of actions is similar to ANI. The implementation of the interpreter in the simulator is based on the implementation in [Rue93] (see section 5.3).

Action semantics was chosen to explore as a semantic framework for modeling languages because it aims to support composable semantic specifications for languages. However, composing modeling languages and their semantic specifications is outside of the scope of this work. In [Doh03] Doh and Mosses succeed in composing a programming language by composing small modules of programming constructs, such as control flow and arithmetic, and their semantic specification with action semantics. This work serves as a first step towards a similar system for modeling languages.

The semantics for action semantics itself are specified using (modular) structural operational semantics [Plo81] by Mosses [Mos92, Mos99]. In [Wan97] action semantics is synthesized with modular monadic semantics [Esp95]. In [Gay02] monadic semantics are also used to create modular semantic specifications. This work only uses the semantics of action semantics for the implementation in the simulator. For other usages of semantic specifications with action semantics both the structural operational semantics and modular monadic mappings should be considered.

Other semantic formalisms for modeling languages
The semantic formalism of action semantics is characterized by the mapping from an input model into an action tree. This input model is static and is not part of the action tree. Any elements of a model that are used will need to be explicitly mapped into the bindings and storage. Graph transformations [Bar02, Hec06] takes a different approach. A model is represented as a graph and modified by applying graph transformation rules on the graph. Where action semantics provides a run time model with transients, bindings and storage, using graph transformations these constructs need to be encoded within the graph. Action semantics also focuses on modularity and composability of semantic specifications, whilst graph transformations does not.

In [Rom07] graph transformations are used as to specify the semantics for the production line language DSML (see the case study in section 6.3). The production line language has a graphical syntax, which is a graph (see Fig 6.3.2). The semantics for the modeling language is defined by creating graph transformation rules. When more than one graph transformation rule is applicable, one is chosen randomly. In our approach we explicitly create a similar approach, by creating a loop with action semantics by checking and enacting rules that use the mapping in the bindings and storage. [Rom07] translates the graph transformation approach into the term rewriting system Maude [Cla07]. Using

Maude the semantic specification are used for verification and validation. Our work does neither verification nor validation and mapping action semantics onto other formalisms is not pursued.

As said, the semantics for action semantics itself are defined by Mosses with structural operational semantics (SOS). In [Wol09] structural operational semantics is used to specify the semantics of modeling languages. That work faced the same difficulties between programming and modeling languages encountered in this work, particularly the usage of abstract syntax trees versus models (graphs). SOS has labeled transitioning systems as its semantic domain. The nodes are graph representations of models and the transitions are proof trees constructed from rewriting rules. Mosses created action semantics after finding it difficult to work with SOS, in this regard our approach provides the same improvements for semantic specifications for modeling languages.

7.5 Discussion

Here we discuss our views on using action semantics as a semantic framework to specify the semantics of modeling language created with MDE. The most important constructs and concepts from action semantics and their application within MDE discussed are: semantic equations, run time model, action notation, action semantics for MDE, and modeling languages and semantics. Finally, some thoughts on the development of modeling languages in general are discussed.
Semantic Equations
The adaptation of semantic equations to work with metamodels worked out well. Specifying the semantics for a programming language or modeling language requires a mapping from the definition of a language to its meaning. In action semantics, this mapping is done with semantic equations, which Peter Mosses took from denotational semantics. Semantic equations are adapted to work with metamodels in this work and they work nicely with the concept of a metamodel. The classes in a metamodel are a unit of abstraction and semantic equations associate a class with its semantics at the same level of abstraction.

Run time model
Action semantic has an explicit run time model consisting of transients, bindings and storage. In all of the examples in this thesis a run time and memory model is required. Having a run time model defined by the semantic framework was therefore a nice property of action semantics. It prevented the need to create an ad-hoc encoding for such constructs, which was exactly what Peter Mosses tried to avoid. In particular the notion of transients, which allows information to be passed between actions, turned out be very useful. It is similar to the construct in programming language of return values of functions, that allow for procedural abstraction and with it a way to structure the semantic specifications into smaller and contained parts. The production line language shows that the run time model can benefit from a way to store and manipulate elements of models in the storage. It would follow the spirit of action semantics if new actions and yielders would be created for this purpose, rather than encoding the constructs with the existing actions and yielders.

Action notation
Action notation in action semantics are actions, data and yielders and the human readable syntax for them. For readability, Peter Mosses created the notation of nested actions by using trees with the root on the left and branching upwards and downwards using vertical bars to indicate the branches (see for example the action tree in Fig 3.3.3 and Fig 6.2.7). While this notation is nice to read and understand, it is not written in the same way, because there is currently no editor to support writing in this manner. This is also true for the syntax that was created for the action semantics metamodel. This discrepancy between the displaying and writing action specifications hinders usability. The English like syntax for action semantics was chosen by Mosses to allow action specifications to be intuitively understood. Actions with names such as *before* and *give* are easy to understand, however, actions such as *furthermore* and *thence* certainly will not immediately convey their proper meaning. This is of course true for any language or notation and action semantics is no exception. But this does tamper the human readability claim.

Action Semantics for MDE
Action semantics for MDE is a framework for the formal specification of modeling languages. Within this framework action semantics is a domain specific language (DSL) for semantics. While a modeling language aims to be a language at the right level of abstraction in respect to the problem domain, action semantics succeeds in being at the right level of abstraction to specify the semantics of computations. In this respect action semantics and MDE combine nicely.

Modeling languages and Semantics
The problem that initiated this work is the need for a way to specify the semantics of modeling languages. The semantics for a language is a necessity for any language and perhaps this need should be the starting point when deciding to create a modeling language. Currently the creation of a modeling language starts with a blank slate, an empty metamodel and thus no abstract syntax and then also no semantic specifications. This has been the design process of every modeling language in this thesis. However, a lot of these modeling languages shared a lot of concepts and semantics. For most metamodels a way to express simple mathematical expression was recreated. It would be nice if a modeling language could be constructed by composing existing metamodels and their semantic specifications. This requires a way to compose both metamodels and semantic specifications. This is the reason action semantics was chosen to investigate as a semantic framework, as it allows semantic specifications to be composed. Action semantics, modular monadic semantics and modular monadic action semantics all agree on the need for combinators, functions and other functional programming constructs to achieve modularity for semantic specifications [Wan97, Gay02]. For action semantics these constructs proved useful to construct semantic specifications with actions as the smallest pieces. The experiences with these constructs in this work suggest that the same constructs are also suitable for composition of whole semantic specifications.

Notes on the development of modeling languages
This thesis uses a number of modeling languages; the example languages for

which the semantics are defined with action semantics and the action semantics modeling language itself (section 4.3).The following are notes on the experiences of developing modeling languages. First, most of the example modeling language shared a common subset of the language that models simple arithmetic. This underlines the need for reusable and composable parts of modeling languages (see previous section).
Second, the tool support for building modeling languages, provided by Ecore and Eclipse, is not well suited for the development of modeling languages. The generated editors and other support by Ecore and Eclipse work well for modeling languages in their final state. However, they are very cumbersome to use during the evolution and iterative development of a language. A change in a metamodel require editors and artifacts to be regenerated and restarted, which is a nuance in any development process.
Action semantics for MDE was prototyped as an embedded DSL because of the lack of reusable building blocks for modeling languages and the poor tool support by Ecore and Eclipse. The host language for this DSL is the Clojure [Clj] language, a Lisp for the JVM, with direct interoperability with the Java language and libraries. Because it is a Lisp, it has support to work with abstract and concrete syntax in the form of s-expression in the language with meta-programming facilities such as macros. Secondly, the benefit of embedding a DSL in a host language is the availabilities of the constructs within the host language. For the Clojure language these include not just arithmetic and function abstractions, but also a collections framework including lazy sequences. These are abstractions and features which can be helpful for any modeling language, regardless of its domain.
Finally, all of this is contained within one language, which supports dynamic and iterative development with its editor, read-eval-print-loop and debugging support. Reusable common abstractions, embedded language support with s-expression and editor support allow the creator of a modeling language to focus on building a language to solve problems in a domain with an iterative, exploratory development process. All of this is available when parenthesis are preferred over angle-brackets.

Current state of this work
The current state of action semantics for MDE allows for the following usages. The semantics for a modeling language, with an Ecore metamodel, can be specified by creating semantic equations. These semantic equations are built with the action semantics modeling language, which has the action semantics metamodel (see section 4.3). The compiler (section 4.4) compiles the semantic specification for a model, in the form of an action tree. Action trees also conform to the action semantics metamodel and are also Ecore files. The simulator (section 4.4) can execute the action trees, either visually by stepping through the action tree or on the command line showing only the final state. With this work the semantics for a modeling language can be specified and models can be prototyped.

7.6 Future Work

This section will outline enhancements and improvements to the current work and further extensions upon this work. This thesis explored the use of action semantics with MDE by adapting action semantics from its origin in programming languages. As outlined previously in this conclusion this adaptation has been successful. Future work for action semantics for MDE includes extending action semantics with a facet for storing parts of a model, composing semantic specifications, extended tool support and using action semantic specification for usages other than prototyping and simulation.

Facet for model storage and modification
The action notations and facets from action semantics that are used in this work are the facets, actions, data and yielders that are used in action semantics as defined by Mosses. These existing facets proved sufficient to be able to specify the semantics of modeling languages. The production line language case study from section 6.3 shows that a way to store elements of a model in memory is desired. This can be achieved by creating actions and yielders that can store and manipulate elements of a model in the storage. Such enhancement should be contained within a facet, which is the way Mosses categorizes actions, data and yielders (see section 3.7). Action semantics aimed to be pragmatic by providing constructs which are abstractions of common constructs in programming languages. A facet with actions and yielders specifically for models and modeling languages will make action semantics for MDE more pragmatic. Such a new facet will provide first class entities to use elements of a model in the action semantics run-time. This is similar to the first class entities that exist in action semantics to enable working with variables in programming languages and their mapping in memory and bindings.

Inheritance and Semantic Equations
This work currently only allows semantic equations over concrete classes (see section 5.5). However, as noted, inheritance in a metamodel implies shared semantics for an inheriting class and its super classes. Therefore, it would make sense to be able to express semantic equations over abstract or super classes. The production line language case study (section 6.3) shows how the lack of this possibility leads to duplication in semantic equations. An alternative solution to this challenge is therefore desired.

Case studies using I/O and concurrency
The purpose of most modeling languages is to ultimately create executable software. Software will require a way to handle input and output events and possibly concurrency. Action semantics provides facets with actions and yielders for this purpose. These facets have not been used in any of the examples or case studies in this work. Specifying the semantics of modeling languages that include these constructs with action semantics should be explored.

Composable languages and semantics
One of the reason to explore action semantics as a semantic framework to specify the semantics of modeling languages is the usage of action semantics for com-

posable semantic specifications. Creating modeling languages by composition of languages or language fragments is a research topic within MDE and this would require composable semantic specifications for these languages as well.

Implement AS for MDE by following Modular Monadic Action Semantics
Extensibility, reusability, modularity and composability of semantics specifications is recurring theme and aim for semantic frameworks. Many semantic frameworks [Esp95,Wan97,Gay02] use monads and monad transformers [Wad90] to achieve these aims. Monads are not used explicitly in action semantics, although action semantics uses related concepts in the form of combinators and actions as functions. The synthesis of action semantics and modular monadic semantics in modular monadic action semantics [Wan97] shows how action semantics can be mapped onto and implemented by using monads. The implementation of the subset of action semantics in this work is rather monolithic. An implementation based on [Wan97] can retain the same syntax and readability of action semantics, while the underlying implementation is more suitable to extensions. This should work well with the other suggestions of future work, such as the creation of a new facet for storing elements of models.

Traces in semantic specifications
An action tree is the semantic specification for a model. This action tree is the result of the application of semantic equations to the model. An action tree has no traces back to the model and semantic equations it originated from. Having such traces would aid the development and debugging of semantic specifications for modeling languages. The traces should include the origin of each action in the action tree by showing the element from the input model and the semantic equation that was applied to it. If the modeling language has a concrete syntax, a trace to lines in the input file should also be included.

Extension of compiler and simulator with language interpreters
The tool support that is provided with this work includes a compiler and a simulator. The compiler compiles the semantic specification of a model into an action tree by using the semantic equations defined for the modeling language. The simulator can execute action trees, both visually and at the command line. Other possible approaches include generating compilers and interpreters for modeling languages. [Wat09] contains an overview of existing tools that generate compilers and interpreters from action semantic descriptions and semantic equations for programming languages. Any such tool for action semantics for MDE will need to address the issue of possible cycles in models, which is handled by the compiler provided with this work (see section 5.3).

Using Action Semantics for verification and validation
The specifications with action semantics are formal and therefore suitable for reasoning and documentation. Within this work the semantic specifications for modeling languages are only used for prototyping by using the simulator. The semantic specifications can also be used for verification and validation, such as reachability and model checking analysis. Using action semantics for MDE for these purposes, will rely on existing approaches for action semantics and its underlying theory of structural operation semantics as defined by Mosses or modular monadic semantics from [Wan97].

References

[Bar02] Baresi, L., Heckel, R., Tutorial introduction to graph transformation: A software engineering perspective. In Proc. Graph Transformation - First International Conference, Barcelona, Spain, 2002

[Bez05] Bézivin, J., On the unification power of models, Software and Systems Modeling, Volume 4-2, pp. 171 - 188, 2005

[Bro92] Brown, D.F., de Moura, H.P., Watt, D.A., Actress: an action semantics directed compiler generator. LNCS, vol. 641, pp. 95109. Springer, 1992

[Che05] Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson E., Semantic anchoring with model transformations. Springer LNCS, 2005

[Cla07] Clavel, M., Duran, F. et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350, Springer, Heidelberg, 2007

[Clj] Clojure, http://www.clojure.org

[Deu00] Van Deursen, A., Klint, P., and Visser, J., Domain-specific languages: an annotated bibliography. SIGPLAN Not. 35, 6, pp. 26-36, 2000

[Doh93] Doh, K.G., Schmidt, D.A., Action semantics-directed prototyping, Computer Languages, 19-4, pp. 213-233, 1993

[Doh03] Doh, K.G., Mosses, P.D., Composing programming languages by combining action-semantics modules, Science of Computer Programming, 47-1 pp. 3-36, 2003

[Eco] Ecore, http://www.eclipse.org/modeling/emf/?project=emf

[Esp95] David Espinosa. Semantic Lego. PhD thesis, Columbia University, 1995

[Fal98] Falkenberg, E., et al. A Framework of Information Systems Concepts, The FRISCO report. 1998

[Gay02] Gayo, J.E.L., Reusable semantic specications of programming languages, SBLP 2002 - VI Brazilian Symposium on Programming Languages, 2002

[Har04] Harel, D., Rumpe. B., Meaningful modeling: Whats the semantic of "semantics"?, Springer LNCS, 2004

[Hec06] Heckel, R., Graph transformation in a nutshell. Electr. Notes Theor. Comput. Sci, 148(1), pp. 187198, 2006

[Hoa69] Hoare, C.A.R., An axiomatic basis for computer programming. Communications of the ACM 12-10, 1969

[Ken02] Kent, S., Model Driven Engineering. In Proceedings of IFM2002, LNCS 2335, Springer, 2002

[Kur05] Kurtev, I. Adaptability of Model Transformations, PhD Thesis, University of Twente, 2005

[Laa09] Laarman, A.W.: An Ontology Based Metalanguage with Explicit Instantiation. Masters thesis, University of Twente, 2009

[Lar06] de Lara, J., Vangheluwe, H., Defining visual notations and their manipulation through meta-modelling and graph transformation. Journal of Visual Languages and Computing 15, pp. 309-330, 2006

[Las00] Lassen, S., Mosses, P.D., Watt, D.A.: an introduction to AN-2: the proposed new version of action notation. In Mosses, P.D., Moura, H. (eds.) Proceedings of the Third International Workshop on Action Semantics, pp.. 19-36, BRICS, 2000

[Mer05] Mernik, M., Heering, J., and Sloane, A. M., When and how to develop domain-specific languages. ACM Comput. Surv. 37-4, pp. 316-344, 2005

[Mil03] Miller, J., Mukerji, J. MDA Guide Version 1.0.1. Object Management Group, 2003

[Mou93] Moura, H., Action Notation Transformations. PhD thesis, University of Glasgow, 1993

[Mou99] Moura, H.P., Menezes, L.C., The ABACO System - An Algebraic Based Action COmpiler. Lecture Notes In Computer Science, vol. 1548. pp. 527-529 Springer-Verlag, 1999

[Mos89] Mosses, P.D., Unied algebras and action semantics. STACS 1989. LNCS, vol. 349, pp. 1735. Springer, Heidelberg, 1989

[Mos92] Mosses, P.D., Action Semantics, Cambridge Tracts in Theoretical Computer Science 26, Cambridge University Press, 1992

[Mos96] Mosses, P.D.,Theory and practice of action semantics, MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science, volume 1113 of LNCS, pp. 371, Springer-Verlag, 1996

[Mos99] Mosses, P.D., A Modular SOS for Action Notation, In Mosses, P.D., Watt, D.A. (eds.) Proceedings of the Second International Workshop on Action Semantics, BRICS, 1999

[OCL03] Object Constraint Language (OCL). OMG Document 03-10-14, 2003

[OMG03] MDA Guide version 1.0.1. Object Management Group, 2003

[Orb94] Orbeak, P., OASIS: An optimizing action-based compiler generator. In Proceedings of the 5th International Conference on Compiler Construction, Edinburgh, Springer-Verlag, 1994

[Pal92] Palsberg, J., Provable Correct Compiler Generation, PhD thesis, University of Aarhus, 1992

[Plo81] Plotkin, G.D., A structural approach to operational semantics. Technical report, University of Aarhus, 1981

[Rom07] Romero, R., Rivera, J.E., Duran, F., Vallecillo, A., Formal and tool support for model driven engineering with Maude. Journal of Object Technology, 2007

[Rue93] Ruei, R., Slonneger, K., Semantic prototyping: Implementing action semantics in Standard ML, University of Iowa, 1993

[Slo95] Slonneger, K., Kurtz, B., Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach, Addison-Wesley Longman, 1995

[Sto77] Stoy, J.E., Denotational Semantics: The Scott-Strachey approach to programming language theory. The MIT Press series in computer science. 1977

[Tof93] Toft, J.U., Feasibility of using RSL as the specification language for the ANDF formal specification. Technical Report, DDC International A/S, 1993

[Wad90] Wadler, P., Comprehending monads, Proceedings of the 1990 ACM Conference on Lisp and Functional Programmging, pp. 61-78, Nice, France, 1990

[Wan97] Wansbrough, K., A modular monadic action semantics, Master thesis, Dept. of Computer Science, University of Auckland, 1997

[Wat09] Watt, D.A., Action Semantics in Retrospect. In Mosses Festschrift, LNCS 5700, Springer-Verlag, pp 4-20, 2009

[Wol09] Wolterink, T.J.L. Operational Semantics applied to Model Driven Engineering. University of Twente, 2009

Appendix A: Action Semantic Descriptions

Activity Diagram Language

```
01: meaning Diagram [nodes:Node*] =
      denotate execute $nodes

02: execute [Node | Rest] [first:Node, rest:Node*] =
      denotate execute $first

03: execute StartNode [next:Node] =
      denotate execute $next

04: execute Assign [next:Node, e:Expr, varName:String] =
05: | | | denotate evaluate $e
06: | | and
07: | | | | give (cell-sort bound to $varName)
08: | | | otherwise
09: | | | | | allocate a cell
10: | | | | then
11: | | | | | | regive
12: | | | | | and
13: | | | | | | bind $varName (given cell-sort#1)
14: | then
15: | | store (given integer-sort#1) (given cell-sort#2)
16: before
17: | denotate execute $next

18: execute Test [e:Expr, next:Node, alternative:Node] =
19: | denotate evaluate $e
20: then
21: | | | check (given truth-sort#1) is (yield true)
22: | | then
23: | | | denotate execute $next
24: | or
25: | | | check (given truth-sort#1) is (yield false)
26: | | then
27: | | | denotate execute $alternative

28: execute StopNode [] = complete

29: evaluate Var [name:String] =
      give (integer-sort stored in (cell-sort bound to $name))
```

```
30: evaluate Number [value:Int] =
      give (yield $value)

31: evaluate BinaryExp [lhs:Exp, bop:BinaryOp, rhs:Exp] =
32: | | denotate evaluate $lhs
33: | and
34: | | denotate evaluate $rhs
35: then
36: | denotate evaluate $bop

37: evaluate Eq [] =
      give (given integer-sort#1) is (given integer-sort#2)

38: evaluate Plus [] =
      give (sum (given integer-sort#1) (given integer-sort#2))
```

Production Line Language

```
001: meaning Line [machines:Machine*, operator:Operator*, trays:Tray*] =
002: | | | | allocate a cell
003: | | | then
004: | | | | | regive
005: | | | | and
006: | | | | | bind "rules" (given cell-sort#1)
007: | | then
008: | | | store empty-list (given cell-sort#1)
009: | before
010: | | | denotate elaborate $trays
011: | | before
012: | | | | denotate elaborate $machines
013: | | | before
014: | | | | | | give empty-list
015: | | | | | then
016: | | | | | | | denotate elaborate_operable_trays $trays
017: | | | | | | then
018: | | | | | | | bind "Operable_trays" (given list-sort#1)
019: | | | | before
020: | | | | | denotate elaborate $operator
021: before
022: | unfolding 'mainloop
023: | | | furthermore
024: | | | | | | give (list-sort stored in (cell sort bound to "rules"))
025: | | | | | and
026: | | | | | | give empty-list
027: | | | | thence
028: | | | | | unfolding 'guardcheck
029: | | | | | | | | | regive
030: | | | | | | | | and-then
031: | | | | | | | | | | give (head-of (head-of (given list-sort#1)))
032: | | | | | | | | | then
033: | | | | | | | | | | enact (given abstraction-sort#1)
034: | | | | | | | then
035: | | | | | | | | | | | check (given truth-sort#3 is yield true)
036: | | | | | | | | | | and-then
037: | | | | | | | | | | | | give (tail-of (given list-sort#1))
038: | | | | | | | | | | | and-then
039: | | | | | | | | | | | | give (concat-list (given list-sort#2)
                (unit-list (head-of (tail-of (head-of (given list-sort#1))))))
040: | | | | | | | | | otherwise
041: | | | | | | | | | | | give (tail-of (given list-sort#1))
042: | | | | | | | | | | and
043: | | | | | | | | | | | give (given list-sort#2)
044: | | | | | | | | then
045: | | | | | | | | | unfold 'guardcheck
046: | | | | | | otherwise
047: | | | | | | | give (given list-sort#2)
```

```
048: | | thence
049: | | | | | check (given list-sort#1 is empty-list)
050: | | | | then
051: | | | | | complete
052: | | | otherwise
053: | | | | | furthermore (choose (given list-sort#1))
054: | | | | thence
055: | | | | | | enact (given abstraction-sort#1)
056: | | | | | before
057: | | | | | | unfold 'mainloop

058: elaborate [Machine | Rest] [first:Machine, rest:Machine*] =
059: | denotate elaborate $first
060: and
061: | denotate elaborate $rest

062: elaborate [Machine] [first:Machine] =
063:   denotate elaborate $first

064: elaborate [Machine Empty] [] =
065:   complete

066: elaborate HeadGen [id:String, counter:int, min:Tray, mout:Tray] =
067: | | allocate a cell
068: | then
069: | | | store (yield $counter) (given cell-sort#1)
070: | | and
071: | | | bind ($id & "_counter") (given cell-sort#1))
072: before
073: | | | give (cell-sort bound to "rules")
074: | | then
075: | | | | regive
076: | | | and
077: | | | | | give (list-sort stored in (given cell-sort#1))
078: | | | | and
079: | | | | | | | denotate elaborate_guard $this
080: | | | | | | and
081: | | | | | | | denotate elaborate_action $this
082: | | | | | then
083: | | | | | | give (concat-list (unit-list (given abstraction-sort#1))
                                 (unit-list (given abstraction-sort#2)))
084: | then
085: | | store (concat-list (given list-sort#2)
                          (unit-list (given list-sort#3))) (given cell-sort#1)
```

```
090: elaborate_guard HeadGen [id:String, counter:int, min:Tray, mout:Tray] =
091: give (closure of (abstraction of (                                  )))
092:     | | | give (integer-sort stored in
                                   (cell-sort bound to ($id & "_counter")))
093:     | | then
094:     | | | give (greater-than (given integer-sort#1) (yield 0))
095:     | and
096:     | | | give (integer-sort stored in
                               (cell-sort bound to ($mout.id & "_nelems")))
097:     | | then
098:     | | | give (less-than (given integer-sort#1) (yield $mout.capacity))
099:     then
100:     | (give (both (given truth-sort#1) (given truth-sort#2)))))))

101: elaborate_action HeadGen [id:String, counter:int, min:Tray, mout:Tray] =
101: give (closure of (abstraction of (                                   )))
102:     | | | give (cell-sort bound to ($mout.id & "_contents"))
103:     | | then
104:     | | | | regive
105:     | | | and
106:     | | | | | give (list-sort stored in (given cell-sort#1))
107:     | | | | then
108:     | | | | | give (concat-list (given list-sort#1)
                                                  (unit-list (yield "head")))
109:     | then
110:     | | store (given list-sort#2) (given cell-sort#1)
111:     and
112:     | | | | give (cell-sort bound to ($mout.id & "_nelems"))
113:     | | | then
114:     | | | | | regive
115:     | | | | and
116:     | | | | | | give (integer-sort stored in (given cell-sort#1))
117:     | | | | | then
118:     | | | | | | give (sum (given integer-sort#1) (yield 1))
119:     | | then
120:     | | | store (given integer-sort#2) (given cell-sort#1)
121:     | and
122:     | | | | give (cell-sort bound to ($id & "_counter"))
123:     | | | then
124:     | | | | | regive
125:     | | | | and
126:     | | | | | | give (integer-sort stored in (given cell-sort#1))
127:     | | | | | then
128:     | | | | | | give (sum (given integer-sort#1) (yield -1))
129:     | | then
130:     | | | store (given integer-sort#2) (given cell-sort#1)
```

```
131: elaborate HandleGen [id:String, counter:int, min:Tray, mout:Tray] =
132: | | allocate a cell
133: | then
134: | | | store (yield $counter) (given cell-sort#1)
135: | | and
136: | | | bind ($id & "_counter") (given cell-sort#1))
137: before
138: | | | give (cell-sort bound to "rules")
139: | | then
140: | | | | regive
141: | | | and
142: | | | | | give (list-sort stored in (given cell-sort#1))
143: | | | | and
144: | | | | | | | denotate elaborate_guard $this
145: | | | | | | and
146: | | | | | | | denotate elaborate_action $this
147: | | | | | then
148: | | | | | | give (concat-list (unit-list (given abstraction-sort#1))
                               (unit-list (given abstraction-sort#2)))
149: | then
150: | | store (concat-list (given list-sort#2)
                    (unit-list (given list-sort#3))) (given cell-sort#1)

151: elaborate_guard HandleGen [id:String, counter:int, min:Tray, mout:Tray] =
152: give (closure of (abstraction of (                                    )))
153:     | | | give (integer-sort stored in
                                  (cell-sort bound to ($id & "_counter")))
154:     | | then
155:     | | | give (greater-than (given integer-sort#1) (yield 0))
156:     | and
157:     | | | give (integer-sort stored in
                              (cell-sort bound to ($mout.id & "_nelems")))
158:     | | then
159:     | | | give (less-than (given integer-sort#1) (yield $mout.capacity))
160:     then
161:     | (give (both (given truth-sort#1) (given truth-sort#2)))))))
```

```
162: elaborate_action HandleGen [id:String, counter:int, min:Tray, mout:Tray] =
163: give (closure of (abstraction of (                                    )))
164:      | | | give (cell-sort bound to ($mout.id & "_contents"))
165:      | | then
166:      | | | | regive
167:      | | | and
168:      | | | | | give (list-sort stored in (given cell-sort#1))
169:      | | | | then
170:      | | | | | give (concat-list (given list-sort#1)
                                                (unit-list (yield "handle")))
171:      | then
172:      | | store (given list-sort#2) (given cell-sort#1)
173:      and
174:      | | | | give (cell-sort bound to ($mout.id & "_nelems"))
175:      | | | then
176:      | | | | | regive
177:      | | | | and
178:      | | | | | | give (integer-sort stored in (given cell-sort#1))
179:      | | | | | then
180:      | | | | | | give (sum (given integer-sort#1) (yield 1))
181:      | | then
182:      | | | store (given integer-sort#2) (given cell-sort#1)
183:      | and
184:      | | | | give (cell-sort bound to ($id & "_counter"))
185:      | | | then
186:      | | | | | regive
187:      | | | | and
188:      | | | | | | give (integer-sort stored in (given cell-sort#1))
189:      | | | | | then
190:      | | | | | | give (sum (given integer-sort#1) (yield -1))
191:      | | then
192:      | | | store (given integer-sort#2) (given cell-sort#1)

193: elaborate Assembler [id:String, min:Tray, mout:Tray] =
194: | | give (cell-sort bound to "rules")
195: | then
196: | | | regive
197: | | and
198: | | | | give (list-sort stored in (given cell-sort#1))
199: | | | and
200: | | | | | | denotate elaborate_guard $this
201: | | | | | and
202: | | | | | | denotate elaborate_action $this
203: | | | | then
204: | | | | | give (concat-list (unit-list (given abstraction-sort#1))
                                 (unit-list (given abstraction-sort#2)))
205: then
206: | store (concat-list (given list-sort#2) (unit-list (given list-sort#3)))
             (given cell-sort#1)
```

```
207: elaborate_guard Assembler [id:String, min:Tray, mout:Tray] =
208: give (closure of (abstraction of(                                    )))
209:      | | | give (integer-sort stored in
                                  (cell-sort bound to ($min.id & "_nelems")))
210:      | | then
211:      | | | give (greater-than (given integer-sort#1) (yield 1))
212:      | and
213:      | | | | give (list-sort stored in
                                (cell-sort bound to ($min.id & "_contents")))
214:      | | | then
215:      | | | | | unfolding 'hashead
216:      | | | | | | | | | regive
217:      | | | | | | | | and
218:      | | | | | | | | | give (head-of (given list-sort#1))
219:      | | | | | | | then
220:      | | | | | | | | | | check (given text-sort#2 is yield "head")
221:      | | | | | | | | | then
222:      | | | | | | | | | | give (yield true)
223:      | | | | | | | | otherwise
224:      | | | | | | | | | | give (tail-of (given list-sort#1))
225:      | | | | | | | | | then
226:      | | | | | | | | | | unfold 'hashead
227:      | | | | | | otherwise
228:      | | | | | | | give (yield false)
229:      | | | | and
230:      | | | | | unfolding 'hashandle
231:      | | | | | | | | | regive
232:      | | | | | | | | and
233:      | | | | | | | | | give (head-of (given list-sort#1))
234:      | | | | | | | then
235:      | | | | | | | | | | check (given text-sort#2 is yield "handle")
236:      | | | | | | | | | then
237:      | | | | | | | | | | give (yield true)
238:      | | | | | | | | otherwise
239:      | | | | | | | | | | give (tail-of (given list-sort#1))
240:      | | | | | | | | | then
241:      | | | | | | | | | | unfold 'hashandle
242:      | | | | | | otherwise
243:      | | | | | | | give (yield false)
244:      | | and
245:      | | | | give (integer-sort stored in
                               (cell-sort bound to ($mout.id & "_nelems")))
246:      | | | then
247:      | | | | give (less-than (given integer-sort#1) (yield $mout.capacity))
248:      then
249:      | give (both (both (given truth-sort#1) (given truth-sort#2))
                      (both (given truth-sort#3) (given truth-sort#4)))
```

```
250: elaborate_action Assembler [id:String, min:Tray, mout:Tray] =
251: give (closure of (abstraction of(                                 )))
252:      | | | | give (list-sort stored in
                          (cell-sort bound to ($min.id & 253"_contents")))
254:      | | | and
255:      | | | | give empty-list
256:      | | then
257:      | | | | unfolding 'removehead
258:      | | | | | | | | regive
259:      | | | | | | | and
260:      | | | | | | | | give (head-of (given list-sort#1))
261:      | | | | | | then
262:      | | | | | | | | | check (given text-sort#3 is yield "head")
263:      | | | | | | | | and
264:      | | | | | | | | | give (concat-list (given list-sort#2)
                                          (tail-of (given list-sort#1)))
265:      | | | | | | | otherwise
266:      | | | | | | | | | | give (tail-of (given list-sort#1))
267:      | | | | | | | | | and
268:      | | | | | | | | | | give (concat-list (given list-sort#2)
                                          (unit-list (given text-sort#3)))
269:      | | | | | | | | then
270:      | | | | | | | | | unfold 'removehead
271:      | | | | | otherwise
272:      | | | | | | give (given list-sort#2)
273:      | | | then
274:      | | | | store (given list-sort#1)
                       (cell-sort bound to ($min.id & "_contents"))
275:      | then
276:      | | | | give (list-sort stored in
                            (cell-sort bound to ($min.id & "_contents")))
277:      | | | and
278:      | | | | give empty-list
279:      | | then
280:      | | | | unfolding 'removehandle
281:      | | | | | | | | regive
282:      | | | | | | | and
283:      | | | | | | | | give (head-of (given list-sort#1))
284:      | | | | | | then
285:      | | | | | | | | | check (given text-sort#3 is yield "handle")
286:      | | | | | | | | and
287:      | | | | | | | | | give (concat-list (given list-sort#2)
                                          (tail-of (given list-sort#1)))
288:      | | | | | | | otherwise
289:      | | | | | | | | | | give (tail-of (given list-sort#1))
290:      | | | | | | | | | and-then
291:      | | | | | | | | | | give (concat-list (given list-sort#2)
                                        (unit-list (given text-sort#3)))
292:      | | | | | | | | then
293:      | | | | | | | | | unfold 'removehandle
```

```
294:     | | | | | otherwise
295:     | | | | | | give (given list-sort#2)
296:     | | | then
297:     | | | | store (given list-sort#1)
                       (cell-sort bound to ($min.id & "_contents"))
298:     then
299:     | | | | give (cell-sort bound to ($min.id & "_nelems"))
300:     | | | then
301:     | | | | | regive
302:     | | | | and
303:     | | | | | | give (integer-sort stored in (given cell-sort#1))
304:     | | | | | then
305:     | | | | | | give (sum (given integer-sort#1) (yield -2))
306:     | | then
307:     | | | store (given integer-sort#2) (given cell-sort#1)
308:     | and
309:     | | | | | give (cell-sort bound to ($mout.id & "_contents"))
310:     | | | | then
311:     | | | | | | regive
312:     | | | | | and
313:     | | | | | | | give (list-sort stored in (given cell-sort#1))
314:     | | | | | | then
315:     | | | | | | | give (concat-list (given list-sort#1)
                                        (unit-list (yield "hammer")))
316:     | | | then
317:     | | | | store (given list-sort#2) (given cell-sort#1)
318:     | | and
319:     | | | | | give (cell-sort bound to ($mout.id & "_nelems"))
320:     | | | | then
321:     | | | | | | regive
322:     | | | | | and
323:     | | | | | | | give (integer-sort stored in (given cell-sort#1))
324:     | | | | | | then
325:     | | | | | | | give (sum (given integer-sort#1) (yield 1))
326:     | | | then
327:     | | | | store (given integer-sort#2) (given cell-sort#1)

328: elaborate [Tray | Rest] [first:Tray, rest:Tray*] =
329: | denotate elaborate $first
330: and
331: | denotate elaborate $rest

332: elaborate [Tray] [first Tray] =
333:   denotate elaborate $first
```

```
334: elaborate Tray [id:String, nelems:int, parts:Part*] =
335: | | give empty-list
336: | then
337: | | | denotate elaborate $parts
338: | | then
339: | | | | | allocate a cell
340: | | | | and
341: | | | | | regive
342: | | | then
343: | | | | | store (given list-sort#2) (given cell-sort#1))
344: | | | | and
345: | | | | | bind ($id & "_contents") (given cell-sort#1)
346: and
347: | | allocate a cell
348: | then
349: | | | store (yield $nelems) (given cell-sort#1)
350: | | and-then
351: | | | bind ($id & "_nelems") (given cell-sort#1)

352: elaborate [Operator Empty] [] = complete

353: elaborate [Operator] [first:Operator] =
354:   denotate elaborate $first

355: elaborate [Part Empty] [] = regive

356: elaborate [Part] [first:Part] =
357:   denotate elaborate $first

358: elaborate [Part | Rest] [first:Part, rest:Part*] =
359: | denotate elaborate $first
360: then
361: | denotate elaborate $rest

362: elaborate Head [] =
363:   give (concat-list (unit-list (yield "head")) (given list-sort#1))

364: elaborate Handle [] =
365:   give (concat-list (unit-list (yield "handle")) (given list-sort#1))

366: elaborate Hammer [] =
367:   give (concat-list (unit-list (yield "hammer")) (given list-sort#1))
```

```
367: elaborate Operator [id:String, from:Tray, to:Tray] =
368: | | | | | denotate mappings_tray $from
369: | | | | and
370: | | | | | denotate mappings_tray $to
371: | | | then
372: | | | | give (concat-list (unit-list (given list-sort#1))
                             (unit-list (given list-sort#2)))
373: | | and
374: | | | allocate a cell
375: | then
376: | | | store (given list-sort#1) (given cell-sort#2)
377: | | and
378: | | | bind ($id & "_current_trays") (given cell-sort#2)
379: before
380: | | | then (give (cell-sort bound to "rules"))
381: | | then
382: | | | | regive
383: | | | and
384: | | | | | give (list-sort stored in (given cell-sort#1))
385: | | | | and
386: | | | | | | | | denotate elaborate_transfer_guard $this
387: | | | | | | | and
388: | | | | | | | | denotate elaborate_transfer_action $this
389: | | | | | | then
390: | | | | | | | give (concat-list (unit-list (given abstraction-sort#1))
                                   (unit-list (given abstraction-sort#2)))
391: | | | | | and
392: | | | | | | | | denotate elaborate_move_guard $this
393: | | | | | | | and
394: | | | | | | | | denotate elaborate_move_action $this
395: | | | | | | then
396: | | | | | | | give (concat-list (unit-list (given abstraction-sort#1))
                                   (unit-list (given abstraction-sort#2)))
397: | then
398: | | store (concat-list (given list-sort#2)
                          (concat-list (unit-list (given list-sort#3))
               (unit-list (given list-sort#4)))) (given cell-sort#1)
```

```
399: elaborate_transfer_guard Operator [id:String, from:Tray, to:Tray] =
400:  give (closure of (abstraction of (                                  )))
401:     | give (list-sort stored in
                               (cell-sort bound to ($id & "_current_trays")))
402:     then
403:     | | | | give (integer-sort stored in
                           (head-of (tail-of (head-of (given list-sort#1)))))
404:     | | | then
405:     | | | | give (greater-than (given integer-sort#1) (yield 0))
406:     | | and
407:     | | | | | give (integer-sort stored in
                  (head-of (tail-of (head-of (tail-of (given list-sort#1))))))
408:     | | | | and
409:     | | | | | give (head-of (tail-of (tail-of (tail-of
                                  (head-of (tail-of (given list-sort#1)))))))
410:     | | | then
411:     | | | | give (less-than (given integer-sort#1) (given integer-sort#2))
412:     | then
413:     | | give (both (given truth-sort#1) (given truth-sort#2))

414: elaborate_transfer_action Operator [id:String, from:Tray, to:Tray] =
415: give (closure of (abstraction of(                                      )))
416:     | give (list-sort stored in
                                (cell-sort bound to ($id & "_current_trays")))
417:     then
418:     | | | | regive
419:     | | | and
420:     | | | | | give (head-of (tail-of (tail-of (head-of
                                                      (given list-sort#1)))))
421:     | | | | then
422:     | | | | | | | regive
423:     | | | | | | and
424:     | | | | | | | | | give (list-sort stored in (given cell-sort#1))
425:     | | | | | | | | then
426:     | | | | | | | | | | regive
427:     | | | | | | | | | and
428:     | | | | | | | | | | | give empty-list
429:     | | | | | | | | | | and
430:     | | | | | | | | | | | choose (given list-sort#1)
431:     | | | | | | | then
432:     | | | | | | | | unfolding 'removechosen
433:     | | | | | | | | | | | regive
434:     | | | | | | | | | | and
435:     | | | | | | | | | | | give (head-of (given list-sort#1))
436:     | | | | | | | | | then
437:     | | | | | | | | | | | | check (given text-sort#3 is given text-sort#4)
438:     | | | | | | | | | | | and
439:     | | | | | | | | | | | | | give (concat-list (given list-sort#2)
                                                (tail-of (given list-sort#1)))
440:     | | | | | | | | | | | | and
441:     | | | | | | | | | | | | | give (given text-sort#3)
```

```
442:    | | | | | | | | | | otherwise
443:    | | | | | | | | | | | | | give (tail-of (given list-sort#1))
444:    | | | | | | | | | | | | and
445:    | | | | | | | | | | | | | | give (concat-list (given list-sort#2)
                                             (unit-list (given text-sort#4)))
446:    | | | | | | | | | | | | | and
447:    | | | | | | | | | | | | | | give (given text-sort#3)
448:    | | | | | | | | | | | then
449:    | | | | | | | | | | | | unfold 'removechosen
450:    | | | | | then
451:    | | | | | | | store (given list-sort#2) (given cell-sort#1)
452:    | | | | | | and
453:    | | | | | | | give (given text-sort#3)
454:    | | then
455:    | | | | | give (head-of (tail-of (tail-of (head-of
                                            (tail-of (given list-sort#1))))))
456:    | | | | and
457:    | | | | | give (given text-sort#2)
458:    | | | then
459:    | | | | | | give (given cell-sort#1)
460:    | | | | | and
461:    | | | | | | | give (list-sort stored in (given cell-sort#1))
462:    | | | | | | and
463:    | | | | | | | give (unit-list (given text-sort#2))
464:    | | | | then
465:    | | | | | store (concat-list (given list-sort#2) (given list-sort#3))
                      (given cell-sort#1)
466:    | and
467:    | | | | | give (head-of (tail-of (head-of (given list-sort#1))))
468:    | | | | then
469:    | | | | | | regive
470:    | | | | | and
471:    | | | | | | | give (integer-sort stored in (given cell-sort#1))
472:    | | | | | | then
473:    | | | | | | | give (sum (given integer-sort#1) (yield -1))
474:    | | | then
475:    | | | | store (given integer-sort#2) (given cell-sort#1)
476:    | | and
477:    | | | | | give (head-of (tail-of (head-of (tail-of
                                                (given list-sort#1)))))
478:    | | | | then
479:    | | | | | | regive
480:    | | | | | and
481:    | | | | | | | give (integer-sort stored in (given cell-sort#1))
482:    | | | | | | then
483:    | | | | | | | give (sum (given integer-sort#1) (yield 1))
484:    | | | then
485:    | | | | store (given integer-sort#2) (given cell-sort#1)
```

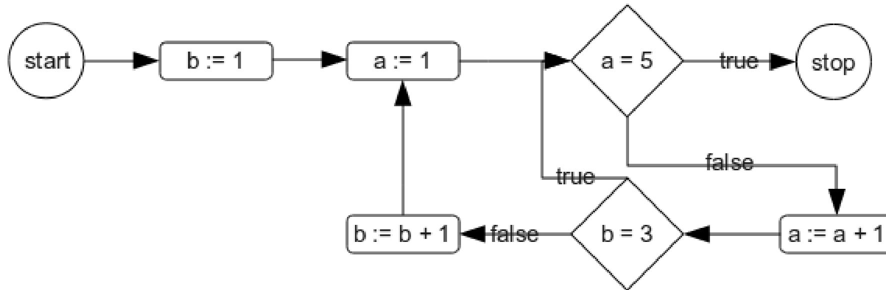
```
486: elaborate_move_guard Operator [id:String, from:Tray, to:Tray] =
487: give (closure of (abstraction of(                                     )))
488:     | give (list-sort bound to "Operable_trays")
489:     then
490:     | unfolding 'hastarget
491:     | | | | | regive
492:     | | | | and
493:     | | | | | | give (head-of (given list-sort#1))
494:     | | | | | then
495:     | | | | | | | | give (not ((given list-sort#1) is
        (list-sort stored in (cell-sort bound to ($id & "_current_trays")))))
496:     | | | | | | | and
497:     | | | | | | | | | give (integer-sort stored in
                      (head-of (tail-of (head-of (given list-sort#1)))))
498:     | | | | | | | | then
499:     | | | | | | | | | give (greater-than (given integer-sort#1)
                                              (yield 0))
500:     | | | | | | | | and
501:     | | | | | | | | | | give (integer-sort stored in
              (head-of (tail-of (head-of (tail-of (given list-sort#1))))))
502:     | | | | | | | | | and
503:     | | | | | | | | | | give (head-of (tail-of (tail-of
                    (tail-of (head-of (tail-of (given list-sort#1)))))))
504:     | | | | | | | | then
505:     | | | | | | | | | give (less-than (given integer-sort#1)
                                          (given integer-sort#2))
506:     | | | | | | then
507:     | | | | | | | give (both (given truth-sort#1)
                       (both (given truth-sort#2) (given truth-sort#3)))
508:     | | | then
509:     | | | | | | check (given truth-sort#2) is (yield true)
510:     | | | | | and
511:     | | | | | | give (yield true)
512:     | | | | otherwise
513:     | | | | | | give (tail-of (given list-sort#1))
514:     | | | | | then
515:     | | | | | | unfold 'hastarget
516:     | | otherwise
517:     | | | give (yield false)
```

```
518: elaborate_move_action Operator [id:String, from:Tray, to:Tray] =
519: give (closure of (abstraction of(                                   )))
520:     | | | give (list-sort bound to "Operable_trays")
521:     | | and
522:     | | | give empty-list
523:     | then
524:     | | unfolding 'findmoves
525:     | | | | | | regive
526:     | | | | | and
527:     | | | | | | | give (head-of (given list-sort#1)))
528:     | | | | | | then
529:     | | | | | | | | | give (not (given list-sort#1) is
        (list-sort stored in (cell-sort bound to ($id & "_current_trays")))))
530:     | | | | | | | | and
531:     | | | | | | | | | | | give (integer-sort stored in
                          (head-of (tail-of (head-of (given list-sort#1)))))
532:     | | | | | | | | | | then
533:     | | | | | | | | | | | give (greater-than (given integer-sort#1)
                                                  (yield 0))
534:     | | | | | | | | | and
535:     | | | | | | | | | | | | give (integer-sort stored in (head-of
                         (tail-of (head-of (tail-of (given list-sort#1))))))
536:     | | | | | | | | | | | and
537:     | | | | | | | | | | | | give (head-of (tail-of (tail-of (tail-of
                                 (head-of (tail-of (given list-sort#1)))))))
538:     | | | | | | | | | | then
539:     | | | | | | | | | | | give (less-than (given integer-sort#1)
                                               (given integer-sort#2))
540:     | | | | | | | then
541:     | | | | | | | | give (both (given truth-sort#1)
                        (both (given truth-sort#2) (given truth-sort#3)))))))
542:     | | | | then
543:     | | | | | | | | check (given truth-sort#3) is (yield true)
544:     | | | | | | | and
545:     | | | | | | | | | give (tail-of (given list-sort#1))
546:     | | | | | | | | and
547:     | | | | | | | | | give (concat-list (given list-sort#2)
                                  (unit-list (head-of (given list-sort#1))))
548:     | | | | | | otherwise
549:     | | | | | | | | give (tail-of (given list-sort#1))
550:     | | | | | | | and
551:     | | | | | | | | give (given list-sort#2))
552:     | | | | | then
553:     | | | | | | unfold 'findmoves
554:     | | | otherwise
555:     | | | | give (given list-sort#2)
556:     then
557:     | | choose (given list-sort#1)
558:     | then
559:     | | store (given list-sort#1)
                   (cell-sort bound to ($id & "_current_trays"))
```

```
560: elaborate_operable_trays [Tray Empty] [] =
561:   regive

562: elaborate_operable_trays [Tray | Rest] [first:Tray, rest:Tray*] =
563: | denotate elaborate_operable_trays $first
564: then
565: | denotate elaborate_operable_trays $rest

566: elaborate_operable_trays [Tray] [first:Tray] =
567:   denotate elaborate_operable_trays $first

568: elaborate_operable_trays Tray [next:Tray] =
569: | | | regive
570: | | and
571: | | | | | denotate mappings_tray $this
572: | | | | and
573: | | | | | denotate mappings_tray $next
574: | | | then
575: | | | | give (concat-list (unit-list (given list-sort#1))
                             (unit-list (given list-sort#2)))
576: | then
577: | | give (concat-list (given list-sort#1)
                         (unit-list (given list-sort#2)))
578: otherwise
579: | regive

580: mappings_tray [Tray Empty] [] = fail

581: mappings_tray Tray [id:String, capacity:int] =
582: | | | | give (yield $id)
583: | | | and
584: | | | | give (cell-sort bound to ($id & "_nelems"))
585: | | then
586: | | | give (concat-list (unit-list (given text-sort#1))
                           (unit-list (given cell-sort#2)))
587: | and
588: | | | | give (cell-sort bound to ($id & "_contents"))
589: | | | and
590: | | | | give (yield $capacity)
591: | | then
592: | | | give (concat-list (unit-list (given cell-sort#1))
                           (unit-list (given integer-sort#2))))))
593: then
594: | give (concat-list (given list-sort#1) (given list-sort#2)))))
```

Appendix B: Action Trees

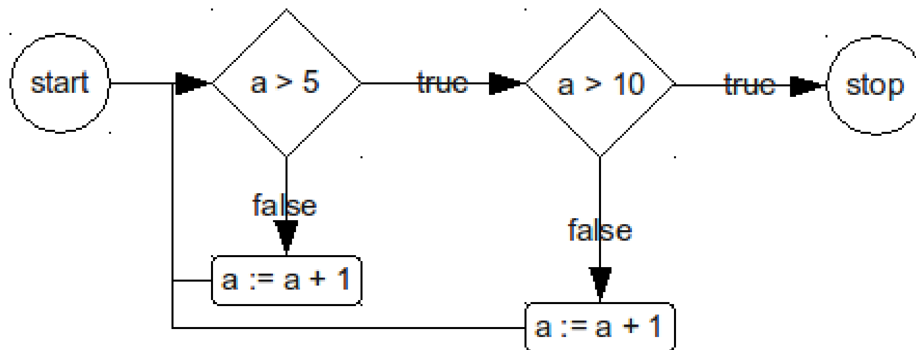
Activity Diagram Language - Nested Loop 1




```
give( yield "1" )
and
give( bound-to(cell-sort, "b") )
otherwise
allocate-a-cell
then
regive
and
bind( "b", given cell-sort#1 )
then
store( given integer-sort#1, given cell-sort#2 )
before
unfolding 'executeAssign13809944
give( yield "1" )
and
give( bound-to(cell-sort, "a") )
otherwise
allocate-a-cell
then
regive
and
bind( "a", given cell-sort#1 )
then
store( given integer-sort#1, given cell-sort#2 )
before
unfolding 'executeTest7739053
give( stored-in( integer-sort, bound-to(cell-sort, "a") ) )
and
give( yield "5" )
then
give( given integer-sort#1 is given integer-sort#2 )
then
```

```
check( yield "true" is given truth-sort#1 )
then
complete
or
check( yield "false" is given truth-sort#1 )
then
give( stored-in( integer-sort, bound-to(cell-sort, "a") ) )
and
give( yield "1" )
then
give( sum( given integer-sort#1, given integer-sort#2 ) )
and
give( bound-to(cell-sort, "a") )
otherwise
allocate-a-cell
then
regive
and
bind( "a", given cell-sort#1 )
then
store( given integer-sort#1, given cell-sort#2 )
before
give( stored-in( integer-sort, bound-to(cell-sort, "b") ) )
and
give( yield "3" )
then
give( given integer-sort#1 is given integer-sort#2 )
then
check( yield "true" is given truth-sort#1 )
then
unfold 'executeTest7739053
or
check( yield "false" is given truth-sort#1 )
then
give( stored-in( integer-sort, bound-to(cell-sort, "b") ) )
and
give( yield "1" )
then
give( sum( given integer-sort#1, given integer-sort#2 ) )
and
give( bound-to(cell-sort, "b") )
otherwise
allocate-a-cell
then
regive
and
bind( "b", given cell-sort#1 )
then
store( given integer-sort#1, given cell-sort#2 )
before
unfold 'executeAssign13809944
```

Activity Diagram Language - Nested Loop 2




```
give( yield "1" )
and
give( bound-to(cell-sort, "a") )
otherwise
allocate-a-cell
then
regive
and
bind( "a", given cell-sort#1 )
then
store( given integer-sort#1, given cell-sort#2 )
before
unfolding 'executeTest1876939
give( stored-in( integer-sort, bound-to(cell-sort, "a") ) )
and
give( yield "5" )
then
give( given integer-sort#1 > given integer-sort#2 )
then
check( yield "true" is given truth-sort#1 )
then
give( stored-in( integer-sort, bound-to(cell-sort, "a") ) )
and
give( yield "10" )
then
give( given integer-sort#1 > given integer-sort#2 )
then
check( yield "true" is given truth-sort#1 )
then
complete
or
```

```
check( yield "false" is given truth-sort#1 )
then
give( stored-in( integer-sort, bound-to(cell-sort, "a") ) )
and
give( yield "1" )
then
give( sum( given integer-sort#1, given integer-sort#2 ) )
and
give( bound-to(cell-sort, "a") )
otherwise
allocate-a-cell
then
regive
and
bind( "a", given cell-sort#1 )
then
store( given integer-sort#1, given cell-sort#2 )
before
unfold 'executeTest1876939
or
check( yield "false" is given truth-sort#1 )
then
give( stored-in( integer-sort, bound-to(cell-sort, "a") ) )
and
give( yield "1" )
then
give( sum( given integer-sort#1, given integer-sort#2 ) )
and
give( bound-to(cell-sort, "a") )
otherwise
allocate-a-cell
then
regive
and
bind( "a", given cell-sort#1 )
then
store( given integer-sort#1, given cell-sort#2 )
before
unfold 'executeTest1876939
```

Appendix C: CD

This thesis is accompanied with a compact disc that contains software, source code, documentation and other files:

1. **Thesis in Digital Form**

   Path /thesis/thesis.pdf

   Description This thesis in digital pdf format.

2. **Tools**

   Path /tools/

   Description The compiler and simulator. Includes compiled versions and all source code files.

3. **Example Modeling Languages**

   Path /examples/

   Description The example modeling languages; the Calculator and Production Line Language. Includes the metamodels, example models and the action semantic descriptions.