Domain specific languages and their type systems

Citation for published version (APA):
Meer, van der, A. P. (2014). *Domain specific languages and their type systems*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven.
https://doi.org/10.6100/IR778421

DOI:
10.6100/IR778421

Document status and date:
Published: 01/01/2014

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Domain Specific Languages and their Type Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op maandag 10 november 2014 om 16:00 uur

door

Arjan Pieter van der Meer

geboren te Voorburg

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. E.H.L. Aarts
1e promotor:	prof.dr. M.G.J. van den Brand
2e promotor:	prof.dr.ir. J.F. Groote
copromotor(en):	dr. A. Serebrenik
leden:	dr. ir. J.P.M. Voeten
	prof. dr. P.D. Mosses(Swansea University)
	prof.dr. M. Mernik(University of Maribor)
adviseur(s):	prof. dr. ir. J.E. Rooda

Domain Specific Languages and their Type Systems

Arjan Pieter van der Meer

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics)

IPA dissertation series 2014-10.

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-3677-1

Printed by the print service of the Eindhoven University of Technology

Cover design based on Wirefox © Alexander Tibus, 2005

Acknowledgements

During the writing and preparation of this thesis, I received the support of many people. I would like to thank in particular:

- my first promotor, Mark van den Brand, for both the opportunity to do this PhD project and for the many discussions and comments through the years.
- my daily supervisor and copromotor, Alexander Serebrenik, for all the advice and innumerable review comments.
- our contact with the CIF developers, Albert Hofkamp, for the helpful discusions and contributions.
- the second Jan Friso Groote, for agreeing to be a member of the PhD committee and for the many helpful comments on this thesis.
- the other members of the PhD committee, Marian Mernik, Peter Mosses, Jeroen Voeten and Koos Rooda, for their helpful comments.
- the fellow PhD students and officemates of the SET group through the years, in no particular order:[1] Marcel van Amstel, Jeroen Arnoldus, John Businge, Yanja Dajsuren, Luc Engelen, Yapin Luo, Maarten Manders, Zvezdan Protic, Ulyana Tikhonova and Bogdan Vasilescu, for their contributions and comradeship.
- my brother and sister, Stefan and Anne Marije, for agreeing to aid me as paranymphs during the defense, and for their support through the years.
- and finally my parents, Jan and Anneke, for their unwavering support.

[1] Well, technically speaking with last names in alphabetical order, but that is still not a particularly *significant* order.

Contents

1 Introduction 1

1.1 Software Construction 1

1.2 Implementation 4

1.3 Problem statement 9

1.4 Research questions 10

1.5 Thesis outline 11

2 Requirements of Type System Specifications 13

2.1 Introduction 13

2.2 Systematic Literature Review of Existing DSL Type Systems . . 14

2.3 Stakeholders 35

2.4 Requirements 37

2.5 Conclusions 38

3 MSOS as Type System Definition Language 41

3.1 Introduction 41

3.2 Preliminaries 45

3.3 MSOS and MSDF: Specifying a Type Checker 48

3.4 Evolution of numerical types in Chi 51

3.5 Typechecking Chi in MSOS 52

3.6 MMT Prototype Implementation 59

3.7 ASF+SDF Prototype Implementation 60

3.8 Related work 65

3.9 Conclusion 66

4 EMF/EMF-TL Introduction 69

4.1 Introduction 69

4.2 EMF 70

4.3 DSL for type system specification 73

4.4 EMF-TL semantics 85

4.5 Implementation of EMF-TL semantics 105

4.6 Related work 110

4.7 Conclusions 113

5 Case studies **115**
5.1 Introduction 115
5.2 Case Study: CIF expressions 118
5.3 Case Study: WebDSL 123
5.4 Case Study: mCRL2 130
5.5 Case Study: POOSL 141
5.6 Conclusion 153

6 Conclusions **157**
6.1 Contributions 157
6.2 Directions for future research 161

A Type system for SLCO **175**

B Type system for CIF **179**

C Type system for WebDSL **207**

D Type system for mCRL2 **219**

E Type system for POOSL **231**

F Summary **247**

G Samenvatting **251**

Chapter 1

Introduction

In order to make use of any computer, we need software to control its operation. One approach to constructing such software is Model Driven Engineering or MDE. In MDE, models created using Domain Specific Languages (DSLs) play a central role. Because they are domain-specific, a large number of DSLs are needed to cover all use cases. Each of these DSLs has its own definition, and needs a separate set of tools to implement it. In this thesis, we will investigate one part of a DSL definition, the type system, and its implementation, the type checker. Our goal is to identify a specification formalism that can be used define type systems in an elegant and clear way, and that can support evolution of both its design and the corresponding tools.

1.1 Software Construction

In many disciplines, computers are used to do tasks that are too large, too complex or too repetitive for humans to perform efficiently. In order for a computer to perform a task, it needs a set of instructions that specify what the task is and how it should be completed. The descriptions of tasks are called *programs*, and a group of programs is collectively known as *software*. The larger the task, the more software is needed to describe it, and the more complex it becomes to construct that software. This increasing complexity means software gets both more expensive and more error-prone. This in turn has led to the development of tools and methodologies intended to reduce the complexity of software during design and development.

Model Driven Engineering

One such methodology for software construction is Model Driven Engineering[1] or MDE [5,15]. The primary observation of MDE is that the larger the software

[1] While the definition of MDE does not necessarily exclude disciplines other than software engineering, that is the primary area where it is used.

system becomes, the more different areas of expertise, also known as domains, it involves. If we try to combine all these different domains into one artifact, the result is, unsurprisingly, large and hard to understand. MDE advocates separating all domains into their own domain models. By clearly defining models and their relations, MDE aims to improve communication about software systems, reuse of components and increase the compatibility of software systems made by different companies or different development teams.

To achieve this, MDE advocates the use of models in every step of the design and development of software. Initially, these models mainly consist of concepts from the domain of the problem that the software is intended to solve. During development, these models are refined by adding more and more concepts from the domain of the implementation, where the problem will be actually solved. In order to do this efficiently, software engineers can create *model transformations*. Model transformations are automated operations that construct new models based on old ones. If we know, for example, how to represent concepts from a problem domain in an implementation domain, we can create a transformation to convert entire models from one domain to another without human intervention.

In order to talk about relations between concepts in different domains, we first need to define those concepts. In particular, if we want to use model transformations, we need to know all concepts that a particular model might contain, in order to ensure that all the information of the model can be transferred to the new model. In practice, this means we need to limit the concepts that can be used in a domain model. A common way to control what concepts can be used in a domain model is to use a *formal language* [55]. Formal languages, as opposed to *natural languages*, are based on a strict structural definition, called a *syntax*. In other words, if something cannot be constructed in accordance to the rules of the language, it simply does not exist as far as the language is concerned. For the so-called *sentences* of the language that can be constructed, we can define specific meaning by creating *semantics* for the language. The semantics of a language can be divided into *static semantics* [113], the meaning of the sentence that can be discovered without executing it, and *dynamic semantics* [113], the meaning of the sentence when executed. Based on intended use, formal languages can be divided into *general purpose languages* [47] and *domain specific languages* [34]. Most programming today is done in general purpose languages or *GPLs*. These languages are designed to be able to handle a wide variety of problems. A consequence of this is that the fundamental concepts they use are not aimed at any particular domain. This can make writing programs harder, because it is not clear how the operations used in the program relate to the problem domain. To a certain extent, this can be improved by using libraries that contain more specific abstractions and vocabulary, but this is still limited by the constructs the GPL provides.

A more in-depth solution to this problem is to create not just a new vocabulary, but an entire new language that matches the domain more closely [100]. These languages are known as Domain-Specific Languages or DSLs. A well-known example of a DSL is SQL [61]. SQL is focussed on accessing and manipulating data stored in structured databases. The data in these databases is grouped into so-called *tables*. Each table consists of *columns* that describe what data is in the table, and *rows*, that contain the actual data. The fundamental nature of tables is reflected in SQL instructions, where we need to describe what tables it applies to, and how these tables will be manipulated in each case. If we want to do something which does not involve tables, this may be impossible or very difficult to achieve in SQL. In contrast, the limited nature of SQL makes it much easier for databases to compute the results of the operations efficiently by using specialized data structures, like indices, or parallelization.

When is a language a Domain Specific Language?

Given that we want to study DSLs, the question arises how we identify DSLs. In other words, what makes a language a DSL as opposed to a GPL? The term is often used without explicit definition, but the following definition was proposed by J. Walton and used in [33][2] and is very similar to the one used in [82]:

Definition 1. *A small, usually declarative, language expressive over the distinguishing characteristics of a set of programs in a particular problem domain.*

Looking at this definition, we note two potential criteria. The first refers to the size of the language, which is defined as small. This size requirement presumably relates to the size of the language definition. Unfortunately, the definition of SQL, as cited earlier, consists of several volumes, while there are several GPLs that have much shorter definitions, e.g. Pascal. Thus, language size alone is not a sufficient criterion. Thus, the difference must lie in the second criterion of the definition: the presence or absence of a target domain which influenced the design of the language. This is not a very clear criterion, because there is no clear definition of what can be considered an application domain. We can try to make it more formal by looking at the ability of the language to express models and computations. This is referred to as its *expressive power*. A DSL should be more powerful than a GPL in some areas and, perhaps more importantly, less powerful in other areas.

Because there are so many different possible computations, and many different ways to describe them, there is no single scale to measure the expressive power of a language. Instead, a common method of expressing language power is to use a comparison [69] to the language corresponding to Turing machines [106]. If a language can describe a Turing machine or an equivalent, it is said to be *Turing Complete* [16]. Note that the implementation of the Turing machine is assumed to have access to infinite memory, in the same way that the Turing machine itself has access to infinite tape. All Turing-complete languages are considered equally powerful, because they can all express any algorithm that a Turing machine can execute. Such a language can rightly be considered general purpose, because it can be used for a wide range of generic problems. This is

[2] Unfortunately, this definition was originally taken from a web page which no longer exists.

gives us a possible criterion to classify languages as DSLs, namely, a language is a DSL only if it is *not* Turing complete.

In practice, however, there are many languages that are regarded as DSLs that fail this criterion. The DSL mentioned earlier, SQL, is an example of this. SQL as originally designed [60] was not Turing complete, but there were many extensions that changed that. Eventually, this influenced the design of SQL to the point that the 2008 version of the standard [61] now describes a Turing complete language even without extensions.

A more precise, but more complex notion of expressive power was defined by Felleisen [42]. In this paper, he suggests comparing two languages by transforming implementations of algorithms from one to the other. If we can do this while maintaining the structure of the implementation, the first language is at least as powerful as the second. If this requires global structural changes to the program, the target language of the transformation is considered less powerful than the source language. Felleissen conjectures that translations from "strong" languages to "weak" languages lead to code duplication, which makes the implementations harder to understand. From the perspective of DSLs, this means that we can consider a DSL more powerful than a GPL in a given domain if translating models to the GPL requires global structural changes. In contrast, we can say that the DSL is weaker than the GPL in other areas if the transformation requires global changes when translating models to the DSL, for example by translating generic constructs into domain-specific equivalents. If we apply this reasoning to SQL, we can see that SQL can represent database operations more succinctly than GPLs, because SQL statements automatically handle all rows in a table and collect the results without needing control-flow constructs. Thus SQL is more powerful than GPLs in that particular area. By making programs shorter, they become easier to design and maintain, as is empirically demonstrated in [72, 73].

Unfortunately, while this criterion is formal, clear and objective, it is also labor-intensive to compare languages in this way. Additionally, in our experience, DSL designers never feel the need to justify their language as domain specific. These two factors mean that this definition is not suitable for use in this thesis. Instead, we are forced to use more subjective criteria. In general, we will instead rely on the judgement of DSL designers to determine if the language they designed is a DSL. For languages that we explore in greater depth, such as those that are the subject of one of our case studies, this is supplemented by our own judgement. In particular, we look for a clear use case where a model constructed in the DSL is significantly more compact than an equivalent model in a GPL would be.

1.2 Implementation

Like any computer language, in order to be practically useable, a DSL needs an *implementation* [44, 50, 81]. A language implementation consists of one or more programs that allow users of the language to create and use models that conform to the specification. These programs can be dedicated to the language, or reused from existing language. The latter approach can save cost, but limits flexibility and is less user friendly [71]. In this thesis, we consider a language implementation as a process with several distinct steps. As noted in [48, 99], not all DSL implementations will feature all steps in equal detail, but all implementations will use them in some way. First, the user creates a model in a textual or graphical form. In case of a textual DSL, the model is then converted to an internal representation in a process known as *parsing*. In graphical DSLs, it is more common for the internal representation to be constructed in parallel with the model, based on the editing actions of the designer. Depending on the language we may then want to refine the model before actually using it. By preprocessing the model in the right way, further steps can be made smaller and more efficient, speeding up both their implementation and their execution. Two common, and closely related, refinements are *scoping* and *typing*, both based on the static semantics of the language. Once all refinements have been completed, we can apply the dynamic semantics. In some languages, this is done directly by combining the model with any required input values to get the output of the model. These languages are known as *interpreted languages*. A disadvantage of this approach is that parsing, scoping, typing and any other model refinement that needs to be done, has to be done again and again each time the model is executed. A way to avoid this is to create a new version of the model that can be executed more easily. This is known as *compilation*. Execution of compiled programs is more efficient than interpretation, but creating the tool that performs compilation, commonly known as the *compiler*, can be complex and expensive. This creates a trade-off, where the cost of creating more advanced tools such as compilers and debuggers is weighed against their benefits.

In the next four sections, we discuss the four main steps, namely, model creation, scoping, typing and execution in more detail. In particular, we discuss how these steps are defined, and how they relate to type systems, which are the main focus of this thesis.

Model creation

When a programmer creates a model, it is often represented as a text. However, not all pieces of text are valid programs or models. Usually, the language designer creates a *grammar* that describes which texts can be used to create valid models. A common way to define a grammar is by using *production rules*. A production rule consists of two patterns, called *left-hand side* and *right-hand side*. A pattern can consist of pieces of actual text, known as *terminals*, and more abstract *non-terminal symbols*. The idea behind a production rule is that in a text, wherever we find an instance of the left-hand side, we can replace it with the right-hand side. To determine if a given text is consistent with a given grammar, we start with a given symbol in the grammar, and try to derive the text by repeated application of production rules. Because texts by definition only contain terminals, this means all non-terminal symbols must be eliminated before this goal can be reached. This process is implemented in a *parser*. If an input piece of text fits in the structure defined by the grammar, the parser constructs a *parse tree* to demonstrate the combination of production rules that led to this conclusion. Depending on the algorithm used to create the tree, a parser can be restricted in the grammars it can handle. A common restriction is that the grammar must be *context-free* [1]. A context-free grammar is a grammar in which the left-hand sides of the production rules are limited to consisting of exactly one single non-terminal symbol. As a result, all production rules that share a symbol in their left-hand side are interchangeable, in that all can result in valid texts if applied. In contrast, in a *context-sensitive* grammar, whether a given pattern is applicable might depend on the presence or absence of other patterns elsewhere in the text. A practical consequence of this is that some common requirements on programs cannot be expressed in context-free grammars, and thus cannot be checked by most parsers. An example is the requirement that a certain piece of text, representing an identifier, must match a piece of text elsewhere. This depends exactly on the context in a way that is not allowed in context-free grammars. Usually, this context-related information is added in the next steps instead.

Binding

In programs and models, it is common that entities are referred to in multiple places. For example, if we create a sufficiently extensive model of a family tree, one person can occur in multiple places in the tree. In fact, given the exponential nature of ancestors needed to avoid duplication, and the limited size of the world population, this must occur in any family tree at some point. In text, the need to represent that one entity exists in multiple places is resolved by using so-called *identifiers*. Identifiers act as names for entities, and multiple instances of an identifier can all refer to the same entity. Less intuitively, the same identifier can act as name for multiple entities. Consider, for example, the number of people who share common first names. Using such an identifier in a model is ambiguous, because we do not know which of the entities is the intended target. An obvious solution is to require that all identifiers correspond to one entity only, but in larger models and programs this is hard and inconvenient to realize.

A common compromise is to introduce *scopes* for identifiers. A scope is a well-defined fragment of a model. We can then use the rule that all instances of an identifier in a scope refer to the same entity. This way, an identifier can be reused throughout a program or model, as long as the scopes do not overlap. The process of identifying the scope an identifier belongs to and what entity corresponds to the identifier in that scope is called *scoping*. Note that scopes are often textually contiguous parts of the program text, but this is not a requirement. Another complication is the concept of *shadowing* [49]. Normally, scopes cannot overlap, because that would lead to ambiguities. If, however, one scope is contained fully in another, we can avoid the ambiguity by stating that only the innermost one applies while the outermost scope is *shadowed*. By doing this, identifiers can be reused for different entities within the same scope.

Another complication are *qualified identifiers*. The idea of qualified identifiers is based on the observation that a programmer may want to use entities that have no identifiers in the current scope. By using a different scope, in which the entity does have an identifier, the desired entity can be identified without adding it to the current scope. A quantified identifier thus consists of two parts, one referencing the scope and one referencing the actual entity within that scope.

Another solution to solve ambiguous identifiers is to use extra information, in addition to the identifier, to decide which entity is referenced by each instance. The most common source of extra information is the type system. In fact, the type system is not only a source of disambiguation information, as many ambiguities are directly related to the kind of information a type system can provide. As mentioned before, the type of an expression provides information on the values it may produce. If the intended target of the identifier uses the results of the expressions in some way, it can be useful to have multiple versions that apply to different types. For example, the compiler might be able to construct more efficient implementations for some types than for others, or some types may require a completely different treatment to achieve the expected result. If the entities in question are functions for manipulating numbers, implementations that require non-negative values can be more efficient than implementations that can also handle negative numbers. If the derived type information allows us to guarantee the function will only be applied to positive values, the compiler can safely select the first implementation. In contrast to a solution where we give each version a different identifier, this use of type information removes the burden of selecting the right version from the programmer. If the compiler uses type information, however, scoping can no longer be completed without consulting the type checker. In the classical approach, this is handled by intertwining the scoping and typing steps, so that each identifier can be resolved when all required information is available. A disadvantage of this is that the definitions of scopes and types are combined, while they are, in principle, separate concepts.

Typing

As mentioned before, a type can be seen as a class of values. In most models, there are many elements that, during execution of the model, manipulate values. Some, usually called *expressions*, even produce new values. Common forms of expressions are literal values, references to variables, collection constructors, unary and binary operations and function calls. We can often use the structure of the model to show that the result of an expression will be of a specific type. A simple example is the expression “2 * 3”. If we interpret the “*” as a multiplication symbol, and “2” and “3” as numbers, this expression will produce a positive integer number, namely 6. In this case, we can compute the value directly, because we have access to all necessary information. If we consider the example “x + 3”, where x is the name of a variable, we cannot compute a value, because we do not know the value x will contain when the expression is evaluated. However, if we know, based on another source, that the value contained in x will be a number, we can still safely predict the multiplication will work correctly. Essentially, we interpret the expression in an abstract way, allowing us to make broad predictions on its results. The process of computing the type of value that will be produced by each expression in a model is called *typing*. During typing, we check if the language elements that receive values can handle values of the computed type. If, for example, we change the previous example to “‘`two`’ * 3”, we try to multiply a word by a number. In most languages, there would be no semantics for such an expression, and trying to execute it would lead to an error. The process of verifying that the types of all expressions in a model are valid is called *type checking*, and the component implementing it is the *type checker*.

Whenever a type checker finds an error, the process of typing can usually not be completed successfully. Instead, an error is reported to the user. In the simplest case, the type checker stops as soon as it discovers an error. This is not considered very user-friendly, so most modern type checkers try to continue until they have attempted to type the whole model. Doing this gives the user a better overview of the overall correctness of the program. It can, however, lead to situations where one error triggers another, leading to a cascade of error messages. Ideally, the type checker should show only the appropriate error messages to the user, filtering those that will go away if the “real” error is fixed. This issue is closely related to the problem of determining what error message to present to the user. For example, suppose the type checker finds an identifier with an associated entity of an incorrect type. We could choose to report this to the user with no further information other than the location of the reference. It might be the case, however, that if the identifier is changed slightly, it becomes correct, or that an entity with the correct identifier exists, but in a different scope. In such cases, it may be more helpful to point this out as a potential solution for the error. It has even been suggested that the type checker should try to make modifications to the model in an attempt to find a version that is correct [75]. Based on this, one or more suggestions to fix the model can be given to the user in addition to the error message. A certain level of care must be taken with this, because textual similarity of two identifiers does not necessarily imply that they are intended to refer to the same element. If they are not, the suggested fix could in fact introduce a new bug into the model.

In modern development environments, typing and type checking are an integral part of the process of creating programs and models. If the system discovers an error during the creation of the model, it will immediately give feedback to the developer, who can then fix the error. From the type checker perspective, this adds the complication that incomplete models must be type checked. Additionally, the type checker should be fast and efficient enough to keep up with the user without interfering with her ability to edit the model. In this scenario, it is very useful if the type checker can operate incrementally, i.e. can type check an incomplete instance of a model, and can reuse those results to type check a more complete version of the same model quicker. Obviously for this to work the parser must be able to produce a parse tree based on an incomplete input text, that additionally can be related to previous and subsequent versions of the same model. If this is not the case, the type checker can either not create partial results to use later, or apply previous results to new models.

Execution

After type checking is finished, and the input text contained no type errors, we now know we have a consistent model. Depending on what the model represents, we may now want to compute some results based on it, refine it further, or create an executable program. To simplify terminology, we will refer to the implementation of these following steps as *execution environments*. Due to the diversity of the potential uses for a model, it is hard to come up with generic criteria that the model created by the type checker should meet. In general, we could say the type checker should compute as much information about the model as possible, because there is more opportunity for reuse if it it is added earlier in the process rather than later. On the other hand, if too much information is inserted into the model, it becomes bloated and more difficult to understand and manipulate properly.

Based on this, it may be more useful to consider the kind of information the type checker should be able to add to the model in order to satisfy the needs of the desired execution environments. We assume that a model consists of model elements with properties. These properties can be references to other model elements, or basic values. Because a type checker must be able to add type information to the model, it should be able to create new model elements that represent those types, and link them to the appropriate elements. In some DSLs, the elements representing the types are already present even in an untyped model, for example as annotations added by the model creator, so the type checker should also be able to link model elements. In order to add information to a model element, it may be necessary to change its structure, so the type checker should also be able to transform elements. In addition to model elements, types may also be represented as basic values, so the type checker should also be able to create and manipulate those.

1.3 Problem statement

The concept of a DSL has been around for some time, but in recent years it has attracted increased interest in connection with MDE, as discussed in greater depth in Section 2.2. With this increased interest comes an increased need to create, maintain and evolve DSLs and their tools. Due to the nature of DSLs, it is important that people who are not experts in the area of language design and construction, or even in computer science, can still contribute to the design of these languages and their tools. To this end, language workbenches [80] and code generators [68] have been developed that allow language designers to specify the desired properties of DSLs and generate implementations based on those specifications. Typically, these so-called *metatools* focus on constructing a DSL model based on input from the user in the form of text or graphical elements, which is then processed further to implement the dynamic semantics of the language. We observe however, that most of these tools pay little attention to the static semantics of the DSL. In our opinion, static analyses like type checking can detect errors and support development for many domain specific languages in the same way they do for general purpose languages, but are different enough from syntax and dynamic semantics to warrant special treatment.

1.4 Research questions

To address the problem stated in Section 1.3, we formulate the central research question that will be the topic of the thesis as:

RQ. *How can DSL type systems be specified in an understandable, formal and evolvable way?*

In the remainder of this section, we will decompose this central research question into more detailed research questions.

As mentioned before, DSLs originate from a wide variety of fields. As a consequence, the syntax and semantics of a DSL can differ substantially from those found in other DSLs or in GPLs. However, it would be impractical to have a separate specialized type system formalism for each DSL. Fortunately, experience from the fields of dynamic semantics and syntax suggests that this is not necessary, because most DSLs consist of variations on several basic features. However, before we can select a formalism that covers these features, we need to identify what they are. Thus, the first research question is:

RQ 1. *What are common features of DSL type systems?*

Another important consideration is that the specification needs to have value for the stakeholders of the language. If the stakeholders cannot understand the specification, or if it provides no benefit for them, we cannot expect them to spend effort on it. Therefore, we need to determine what the views of the stakeholders are on type systems. In particular, the second research question is:

RQ 2. *How can DSL type systems specification help stakeholders reach their goals?*

One aspect of DSLs that is of special interest to us is that of language evolution. Because DSLs are by nature limited in what they can express, they are more likely than GPLs to require evolution in response to domain changes. If this results in, for example, new constructs being introduced, the static semantics and its tooling will have to be updated. Obviously, our specification formalism has to support this evolution. The research question is:

RQ 3. *How can DSL type systems specification assist the process of language evolution?*

Once we have determined what common features of DSL type systems are, we need to find one or more specification formalisms that can express as many of these features as possible in a clear and efficient manner. In practice, this means that for each DSL, a choice must be made between several formalisms that have their own advantages and disadvantages. We observe, however, there is little recent work done in this field. In order to address this for as many potential DSLs as possible, in this thesis, we choose to focus on one specification formalism that can cover the requirements most commonly found in DSLs. Thus, the research question is:

RQ 4. *What is the specification formalism most suited for describing the type systems of DSLs?*

Once we have chosen a suitable formalism for DSL type system specification, we need to make sure that common constructs can indeed be expressed in a concise and flexible way. In order to do this, we have to select a number of DSLs and define their type systems, so that we can see how the formalism works in practice. The corresponding research question is:

RQ 5. *How can DSL type system features best be expressed in our chosen formalism?*

1.5 Thesis outline

This thesis is divided into six main chapters. The first two contain this introduction, Chapter 1, and our discussion the general properties of type systems and type checkers, Chapter 2. The next chapter, Chapter 3 describes our experiments in using MSOS as a type system specification language. We describe how MSOS can be used to define type systems and how a type checker can be generated based on the specification. This chapter is based on [21] and [22].

During this research, we discovered MSOS did not meet our requirements as effectively as we hoped. In order to improve on MSOS, we introduce our own type system specification language, EMF-TL, in Chapter 4. Next, in Chapter 5, we describe case studies for four languages we chose based on our SLR: CIF, WebDSL, mCRL2 and POOSL, where we evaluate the new language.

Finally, the last chapter of this thesis, Chapter 6, contains our overall conclusions and vision on future work.

Chapter 2

Requirements of Type System Specifications

In order to answer some of the research questions stated in Section 1.4, we found we need to know more about the properties of DSL type systems and the desired properties of DSL type system specifications. We address this in this chapter by performing a systematic literature review (SLR) on published DSLs. For this SLR, we collected a large number of papers describing DSLs and used this information to establish which type system properties are common among DSLs. We also look at the various stakeholders involved with DSL type checkers. Using the knowledge gained from the SLR and the stakeholder analysis, we formulate a number of requirements a type system specification should meet.

2.1 Introduction

In our problem statement discussed in Section 1.3, we indicated that we want to develop a language for the specification of type systems, and in particular for the type systems of DSLs. We also indicated that we want to use this language to generate tools, *type checkers*, that implement the described type systems. In order to determine what is required to achieve this, we first need to answer RQ 1 and determine the common properties and components of DSL type systems. To this end, we conducted a systematic literature review, described in Section 2.2. In addition to the type system viewpoint, we also need to determine the desired properties of the generated type checker, because they may influence the design of the language as well. We do this by considering the type checker and its specification from the perspective of several stakeholders (Section 2.3). In Section 2.4 we summarize our findings by providing a list of requirements a type system language should fulfill.

2.2 Systematic Literature Review of Existing DSL Type Systems

One measure of the quality of a type system specification language is the number of practical type systems the language can describe. Ideally, we would like to have sufficient flexibility to define the type system of any possible DSL, while avoiding type system specification language constructs that would be too expensive to implement or inhibit potential analysis of the type systems written in the language. In practice, many DSLs are confidential, poorly documented or very obscure, which makes it hard to sensibly test whether a type system formalism can indeed describe any DSL type system. A more achievable goal is to cover a “typical" DSL, a DSL that uses only type system features that are common to most DSLs. Unfortunately, it is again unclear what this would entail. The nature of DSLs makes them hard to compare, and there is no clear sample set of DSLs that can be used as a basis for the comparison. In order to get a clear picture of what features are common in DSL type systems, we conducted a Systematic Literature Review or SLR [67]. As described in [67], doing an SLR requires a Review Protocol, that defines how this specific review will be conducted. We discuss the elements of the protocol in the next sections.

Research Questions

The first element of the protocol is the definition of the research questions. These research questions refine RQ 1 and are the prime factor that determines what literature will be reviewed, what information we want to gather from each paper, and how we will use that data to achieve the goal of the review. The research questions we have defined are:

RQ 1.1. *How are DSL type systems described?*

RQ 1.1 is related directly to the main purpose of this SLR: to discover how DSL type systems are currently defined. We address this question by looking at the sections discussing semantics of the DSL(s) in each paper.

RQ 1.2. *How complex are DSL type systems?*

RQ 1.2 is connected to the question whether it is necessary to describe the specific static semantics of the DSL in a formal way. If a DSL is very small, its static semantics may be too trivial to be worth formalizing.

RQ 1.3. *What type systems do DSLs have?*

RQ 1.3 is related to the elements common to DSL type systems. We expect that several common properties originating from GPL type systems can also be found in the DSL type systems. Because the level of documentation for DSL type systems can vary greatly, we cannot predict beforehand what properties will occur often enough to be interesting. However, to provide some structure to this question, we have defined a number of subquestions, RQ 1.3.1 to 1.3.4 that we certainly want to answer for each DSL. The first subquestion concerns the position of the type system as either static or dynamic semantics. Static type systems are applied earlier (e.g. during the compilation process) than dynamic type systems which are applied during execution. This distinction strongly affects both the information available to the type system and the actions that can be taken based on the results.

RQ 1.3.1. *Are DSL type systems static?*

The second subquestion is about the guarantees that are made for a correctly-typed model. This property is commonly referred to as the strength of the type system. An early definition of this concept can be found in [77], which states that in a strong language "whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function". In contrast, in a weak type system this requirement is not enforced, or only enforced to a limited extent. It must be noted that a number of variant definitions exist, primarily inspired by discourse on the relative merits of programming languages. In this thesis, we decided to use the definition by Liskov et al. [77], because it is both clear and relevant to all DSLs.

RQ 1.3.2. *Are DSL type systems strong?*

The third subquestion is related to an important feature of some GPLs, namely the existence of special types known as objects. In GPLs, this feature is considered so fundamental that languages that have it are placed in a separate category, namely *object-oriented languages.* We would like to know if DSLs are similarly affected.

RQ 1.3.3. *Are DSL type systems object-oriented?*

The fourth and final question is concerned with how the type system is represented in actual DSL models. In most languages, constructs like declarations require explicit type information, while in others, this is considered unnecessary. The more type information is implicit, the more type inference has to be done as part of the type system to fully type a model.

RQ 1.3.4. *Do DSL type systems feature explicit declarations?*

Search Process

We gathered papers for our SLR in two complementary ways. We observed that the term DSL has not been in common use for quite as long as domain specific languages exist. In other words, DSL existed before they were called such. A number of early DSLs have been recognized and catalogued as such in [34]. Hence, we also choose this article as the point when the term DSL was sufficiently widespread to effectively use search engines. For the period preceding the publication of [34], we include in our SLR the DSLs described in [34]. For the DSLs published after [34], i.e. from 2000 onwards, we used search engines provided by three major scientific publishers: ACM, IEEE and

Springer. While this means not all computer science conferences and journals are covered, we are confident we can acquire a representative sample is this manner, which is the goal of this SLR. For ACM, the search engine we used is called the ACM Digital Library[1], and we performed a search on the search term "`Domain Specific Language`" among author's keywords. For IEEE, the engine is called IEEE Xplore[2], and we used the search term "`"Domain specific language"`", but on index keywords. For Springer, the search engine is called SpringerLink[3]. Because SpringerLink does not support search on keywords, we instead searched for the phrase "Domain Specific Language". We also restricted our search to conference proceedings and journals, to guarantee all materials found were covered by peer review. In all cases, we restricted our search to material published between January 1, 2000, the year the annotated bibliography was published, and December 31, 2012. The ACM and IEEE searches were performed in December 2012, the Springer search was performed in May 2013. This means papers presented in 2012 but not yet published may have been missed. Overall, this will not impact our general results beyond reducing the number of DSLs in our study. In cases where it is more directly relevant, the missing data will be discussed explicitly.

Inclusion Criteria

From the results of the search, we then proceeded to select the papers of interest. We included papers if they met at least one of three criteria:

1. The paper introduces a DSL. An example is [104].

2. The paper introduces a software framework or design method that includes a DSL. An example is [3].

3. The paper describes a DSL construction method using a DSL as a main example. An example is [95].

Exclusion Criteria

After the search, we first collected a number of statistics on the complete set of DSLs. We then performed further analysis on a more limited set of languages, based on the following exclusion criteria.

No type system Papers describing a DSL that is untyped by design will by definition not include a description of a type system.

Internal Internal DSLs, also called embedded DSLs [59], share the type system of their host language. We therefore do not consider type systems of these languages as prime examples of DSL type systems.

[1] http://dl.acm.org/

[2] http://ieeexplore.ieee.org/Xplore/home.jsp

[3] http://link.springer.com/

No information If the information available on the DSL is too superficial, we cannot draw useable conclusions about its type system. Thus, we excluded all language which we rated as quality *none* based on the criteria presented in Section 2.2.

Finally, if a DSL was described in multiple papers, for example because it was updated during the search period, we combined information from all papers available, using the most recent information available in case of contradictions.

Quality Assessment

Not all papers introducing DSLs provide a significant amount of detail about the corresponding type system. In fact, in many cases, the type system is not discussed at all. To some degree, we can use example programs and models to uncover implicit information about the type system. Doing this introduces a judgement factor, and therefore we consider the DSLs where this is necessary of lesser quality. We rate DSLs on a scale that goes from *none*, for DSLs that do not even have an example model, via *basic* (example programs that allow some type system estimation) and *medium* (formal or informal type system description) to *high*, for DSLs for which we have both one or more examples and a description.

Data Collection

From the papers we collected, we extracted several sorts of data. The first category includes metadata on the papers. We extract this to be able to see time-related trends in DSL design, for example due to advances in supporting technology, and the possible differences between DSLs published in different sources.

Paper The papers we used to collect the data on the DSLs. Note that a single DSL can be discussed in multiple papers, and one paper can discuss multiple DSLs.

Authors The authors of the papers.

Years The years in which the papers on the language were published.

Source Was the language described in conference or in journal paper(s)?

Source name The name(s) of the conferences and journals in which the papers appeared.

The next category describes general language properties. We used this data in combination with the exclusion criteria to decide which papers we would analyze further. As such, we refer to Section 2.2 for details on why we choose these particular properties of the DSLs and papers we found for this purpose.

Syntax The kind of syntax used by the DSL. Most languages use either textual or graphical syntax, but some use a combination or allow both.

Internal The degree in which the DSL is connected to a host language.

Typed The presence of a type system in the DSL.

Information The level of information available for the DSL.

The next category describes the various type system properties we are interested in. They are listed in no particular order.

Binary expressions Binary expressions are one of the most basic forms of used language constructs, but some languages, usually oriented at structural rather than behavior description, do lack them. We consider this to be a simple indicator of the potential complexity of the type system, with type systems for languages without binary expressions being simpler than for those that do feature them. We use this category as a metric for RQ 1.2.

Static [113] An important property of a type system is when it is applied. Broadly, type systems can be divided into those that are applied before the model is executed, and those that are applied during execution. The former are referred to as static type systems, the latter as dynamic type systems. This category relates directly to RQ 1.3.1.

Strong [77] The strength of a type system refers to its power to stop execution if it finds type errors. If models can be run despite inconsistencies found by the type system, the type system is considered weak. If a type system prevents execution for models it cannot type, it is considered strong. This category relates directly to RQ 1.3.2

Explicit Most DSLs, especially those with strong type systems, require the programmer to provide information on the intended types in the model. Most commonly, this is done in the form of type annotations for declarations, indicating their intended use. These annotations can provide the type system with important information that enables the detection of more errors. We call a type system explicit if these annotations are required for at least some forms of declaration. This category relates directly to RQ 1.3.4

Overloading [1] As a convenience for programmers, it can be useful to enable the type system to use type information to remove potential ambiguities from a model. A common application of this type system feature is allowing ambiguous calls to functions or procedures, that are separated based on the types of the arguments and/or results. This is referred to as overloading. We use this category as a metric for RQ 1.2

Presence of objects With respect to programming and modeling languages, object orientation refers to a way of organizing definitions of data and behavior in models. One of the core concepts of object-orientation as identified by Pierce [88] is encapsulation. In this SLR, we handle this trait by looking for constructs that define both data and behavior, which we refer to as objects. This category directly relates to RQ 1.3.3.

Inheritance [88] In object-oriented languages, definitions of object types can often be related through a special mechanism called inheritance. This is another of the features identified in [88] as fundamental to object orientation. This category directly relates to RQ 1.3.3.

Inference [88] In type systems without inference, annotations are required for declarations to indicate their intended use. In contrast, type inference is the ability to deduce the type of expressions without referring to explicit annotations. By allowing explicit annotations to be dropped, type inference can reduce the effort required to construct models, at the cost of type system complexity. We use this category as a metric for RQ 1.2.

Parameterized types For complex types, there often exist a number of variations that are very similar. For example, a container type can have variations for different size or different element types. By giving these types parameters, all versions can be represented without needing a large number of predefined types. We use this category as a metric for RQ 1.2.

The final category contains details on specific types that DSLs use.

Boolean Some of the most basic values in languages are the boolean true and false. This column records if the language has an explicit type for boolean values.

Numerical Next to boolean values, numerical values are very common in languages. In contrast with booleans however, there are often several kinds of numbers with different domains. This column records what numerical types the languages have.

String The final common kind of value is textual. These are commonly referred to as strings. Some languages also have a type for individual characters. This column records what textual types the language has.

Other basic types Depending on the domain, the language can have other basic types that are not Boolean, Numerical or String. Examples include graph concepts like nodes and edges, flows and time values. This column records what other basic types the language has.

Collection types Many DSLs have specific constructs to store multiple values together. These are referred to as collections, and common examples are lists and arrays.

Objects Not all object-oriented language refer to objects by the name "object". This column describes what object types exists in the language, if any.

Records Some languages allow complex types to be defined consisting of several parts, each with their own type. We refer to those types as record types, and this column describes their presence and name.

Functions If functions are first-class values in the language, we need function types to describe them. This column records what languages have function types.

Union Union types are constructed by combining two other types, creating one type that contains all values from both component types.

Other Any built-in type that does not fit in any of the other categories is listed here.

Results of the SLR

The full results of the SLR will be available online [79]. Here, we first present an overview of the data we collected. Because of the diverse nature of DSLs, we first focus on a number of general properties that can tell us something about the complexity and maturity of the DSL. Table 2.1 shows the results concerning the kinds of DSLs we discovered during the SLR. In this table, each column contains the results for one property we looked at. For each property, we give the number of languages that have that property, in the `TRUE` row, the number of languages that do not have the property, in the `FALSE` and the number of languages where we were unsure, in the `?` row.

	Syntax					Information		
Category	Textual	Graphical	Combination	Internal	Typed	Discussion	Example	Future work
TRUE	1601	194	16	238	791	318	1563	18
?	24	24	59	5	162	0	34	0
FALSE	178	1585	1728	1560	850	1485	206	1785

Table 2.1: SLR DSL General Properties (N=1803)

One such property is the kind of syntax a DSL uses. As can be seen from the table, the majority of DSLs use textual syntax, but there are a significant number of graphical DSLs as well. Some DSLs use models that consist of both graphical and textual components, and these are listed in the "Combination" column. In our survey, the language WebSpec [25] is an example of such a language. In WebSpec, a graphical language of states and transitions is combined with a textual language describing the actions associated with the transitions. Note also that some DSLs have multiple forms of syntax: this is the reason the number of graphical languages is larger than the number of languages that are not textual. An example is the Autosar [86] language, which has a graphical language primarily intended for model construction, and a textual language intended for model interchange.

The next column deals with internal versus external languages. Recall that a language is called *internal* if it is implemented using the existing infrastructure of another language, generally a GPL. In contrast, *external* languages have an independent set of tools. Making a language internal limits the design of the language, but greatly simplifies the implementation of the language. Nevertheless, we can see that the designers of most DSLs opt to fully implement their languages themselves, instead of expanding on existing language infrastructure. A possible reason might be that designers want to avoid the complexity of GPLs, and an embedded DSL by nature involves a close connection with a GPL. However, more research in this area would be needed to make more definite claims on this topic.

The most important DSL property we considered is whether the DSL is typed. This clearly relates to the topic of this thesis, but is also an indication of the complexity of the language. If a language only deals with very specific data, like numeric values, a type system might not be necessary. In fact, we can see that only just under half of the languages we could classify are typed. In the rest of this chapter, we will ignore untyped languages unless specifically noted.

The last three columns deal with the information available on the language and its type system. This is closely related to our definition of quality as given in Section 2.2. Recall from the exclusion criteria from Section 2.2, the ideal situation is that the paper(s) discussing the DSLs mention the type system explicitly. However, for many DSLs, this is not the case. For those cases, we looked at such examples to get some idea of what features the type system has. Note that in some papers, it was unclear to what extend examples shown are true, complete, examples of the DSL, or only fragments or pseudocode. We marked those as ? to indicate this uncertainty. The information we can get from examples is more limited, and the uncertainties are greater. Finally, we looked whether the type system specification is mentioned in the future work of the papers. We choose to include this column to acknowledge the effort of authors who choose not to include a description of the type system in their paper due to space limitations.

Based on Table 2.1, we selected the DSLs we wanted to investigate further. As mentioned before, untyped languages have no type system that can have properties, so we did not attempt to collect property data on them. This results in a set of 791 DSLs. Secondly, we did not consider internal DSLs, because though these are usually typed, the type system is intertwined closely with the type system of the host language. These, we do not consider true examples of DSL type systems. There are 573 external DSLs in our data set. Finally, if the paper(s) describing the DSL do not feature either an example or a discussion of the type system, there is simply not enough information to use. In fact, in such cases, we can often not determine if the language is typed at all. It may still be possible, for example if the language is explicitly mentioned to be typed, or a metamodel is provided that includes type information. This selection of external languages with type systems with sufficient detail resulted in a set of 528 DSLs.

Table 2.2 shows the type system features for these 528 DSLs we extracted from the papers describing each language. The first feature deals with the indicator we choose for the complexity of a language: binary expressions or their equivalent. The next two columns describe some properties relating to the type system implementation: the way it is applied (statically or dynamically) and its strength. For DSLs, we observe that most are both statically and strongly typed. We must, however, add the caveat to this that both application method and strength are rarely explicitly mentioned in papers discussing DSLs. Thus, for 324 DSLs, the information is estimated based on examples. We worked under the assumption that if a language is compiled and has explicit types, the compiler would use the types during compilation, making the language statically and strongly typed, because incorrect types would lead to incorrect compilation.

Category	Binary Expressions	Static	Strong	Explicit annotations	Parameterized types	Presence of objects	Overloading	Inference	Inheritance
TRUE	469	476	480	438	201	240	104	58	92
?	34	31	34	28	41	23	392	34	161
FALSE	25	21	14	62	285	264	30	435	274

Table 2.2: SLR DSL Type System Characteristics (N=528)

The next property we considered is whether the type system requires the programmer to add explicit type annotations to the model. Because explicit type annotations tend to simplify the implementation of type systems, we expected most DSL type systems to use them, and we observe that this is indeed the case. In this case, however, it should be mentioned that explicit type annotations are a clear sign a language is typed. In contrast, an untyped language will not have any type annotations. Therefore, the presence of type annotations directly influenced our assessment on whether the language was typed, which may have influenced the ratio between explicit and implicit languages in our results.

The next column concerns parameterized types. By parameterized types, we refer to properties that can be set by the user that provide more detail on the kind of values a type represents. For example, a container type may have parameters that describe the type of elements it can contain or its size. A more specific example is the Shape type found in the StreamIt [98] language. The Shape type defines matrices of various sizes and dimensions, like `Shape[10]`, `Shape[15,10]` or `Shape[3,3,3]`. By using parameterized types, a large variety of shapes can be defined without a large number of specific types. From our data, we observe that a majority of DSLs do not have parameterized types, but a significant number do. This is in contrast with the previous characteristics, which were implemented by most DSLs. We expect that many DSL designers felt containers were not needed to represent models in their domain, so they did not include parameterized types either.

The next column deals with the wider of concept of object-orientation. Since object-orientation is such a large concept, we expected it would be difficult to discern if and to what extent a DSL is object-oriented. Instead of deciding whether the language is object-oriented, we decided to consider two simpler properties: presence of objects and inheritance. The first of these is the basic presence of objects. We define objects as types that contain both user-defined data and behavior. From our SLR results, we observe that most DSLs do not have such types, and based on that we conclude most DSLs are not object-oriented. We expect most DSL designers did not find the effort required to add object-orientation worthwhile, or decided that object-orientation did not fit their desired domain constructs.

The next property we considered is overloading. Overloading simplifies model construction, and thus is a useful feature for most DSLs. However, it is also non-trivial to implement, because it means ambiguity is no longer automatically erroneous. An additional complication is that it is hard to judge whether a language supports overloading based purely on examples, which explains the higher number of“unknowns" in the Overloading column in Table 2.2. Despite this, we observe that DSLs do use overloading, but this is based on only a small sample compared to the overall number of languages.

The next property we considered is type inference. Like with some other properties, it is difficult to tell whether a language supports type inference based on examples, so we mainly based this column on type system descriptions. We observe that most languages, roughly 83%, do not use type inference.

The final characteristic is inheritance. This is the second object-orientation related property, and given the limited popularity of objects in general, it is unsurprising that inheritance is the least popular characteristic in our survey. Like with objects, we expect implementation effort is a significant factor here, as well as the generally limited size of DSLs and their models, which makes the increased modularity made possible by inheritance less urgent.

In this section, we describe the results of our SLR of type systems of DSLs. Recall that we collected DSLs by looking at papers published by three major publishers, and a DSL bibliography. For each DSL we found, we attempted to determine general language properties based on the information in the papers. We collected details of 1803 DSLs, of which 791 were typed. Based on this we could already conclude that a significant number of DSLs are typed, meaning there should be a sufficient number of type systems to formalize. To get a better idea of what would be required of a type system formalism, we selected 528 DSLs and collected further information on type system properties. Based on this data, we conclude DSL type systems are often static and strong, while more advanced features like object-oriented, type inference and parameterized types are less common.

Evolution of DSL Type System Properties

The primary conclusion of our SLR is that a significant number of DSLs are typed. We illustrate this and various other statistics with a series of figures starting with Figure 2.1. These figures show how the various characteristics of DSL type systems change over time. In this context, we should recall that we collected the SLR data in December 2012, so our data for 2012 is not necessarily complete. This is indicated in the figure by the underscore under the year 2012. In the first figure, we observe that the typed languages 🟩 outnumber the untyped languages 🟥 in most years. We also observe that the ratio between typed and untyped languages does not fundamentally change over time, staying at roughly 50-50 to 60-40 for typed versus untyped, if we discount the unknowns.

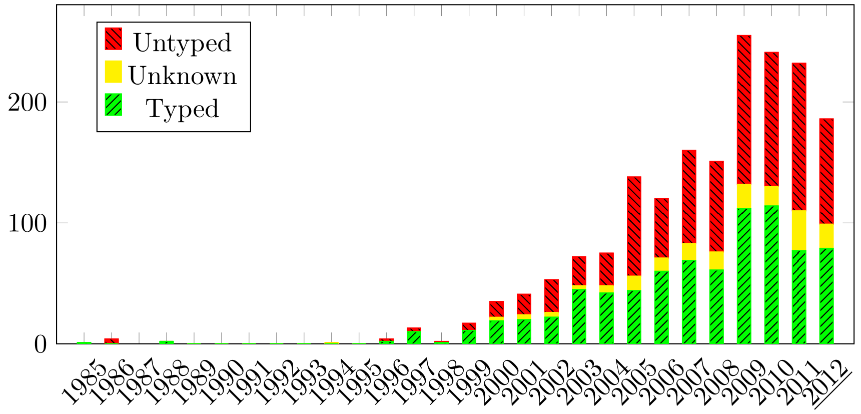


Figure 2.1: Typed DSLs: Roughly half of languages typed. N = 1803.

Given that so many DSLs are untyped, we might ask if this is related to the effort required to build the necessary tools. Though we have not investigated the motivations of the DSL designers, we observe that this would be less of an issue for internal DSLs, because of the support provided by the host language type system. A survey among DSL designers which would clarify this is considered as future work. In Figure 2.2, we show the relation of internal🟩 versus external🟥 DSLs in our study. We observe most DSLs are external. While our data shows internal DSLs are growing more popular in absolute terms, rising from 3 instances found in 2000 to 29 instances in the year 2012, the variation

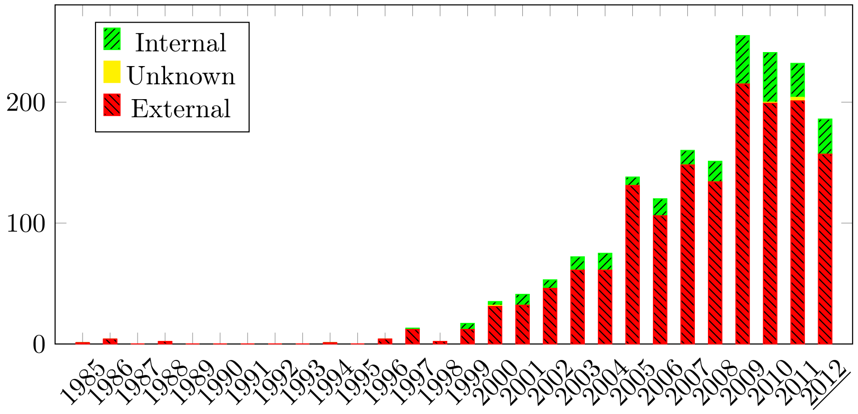


Figure 2.2: DSL Implementation: Most languages are not embedded. N = 1803.

in our data is too large to say if this means the fraction of DSLs that is internal is also rising.

Another factor that could impact the presence of type system is the syntax of DSLs. In Table 2.3, we show that graphical languages are less likely to be typed than textual languages. For the purpose of this table, we only considered external languages. Note that during our data collection, we found 98 languages that have multiple forms of syntax, but we ignored that for the purpose of this table and included them in both counts, because they do not significantly impact the results. From the table, we observe that graphical languages are considerably less likely to be typed than textual languages. If we apply a chi-square test to determine if the typed property is independent of syntax, we find a p-value of less than 0.005, indicating the independence to be very unlikely. We must be careful, however: because graphical languages use constructs that make types less explicit those used in textual languages, we must consider the possibility of a potential bias during the collection of the data, where typed graphical languages are not recognized as such. In order to see if this is the case, we also looked at the presence of type system discussion for graphical and textual languages. Assuming designers of both textual languages and graphical languages are equally likely to present a discussion of a type system if one is present, that gives us a separate measure of how many typed languages there are. In the table, we can see that a quarter of the textual languages have a type system discussion, but less than a tenth of the graphical languages do. Again, if we apply a chi-squared test to determine is the presence of discussion is independent of syntax, we find a p-value of less than 0.005, indicating it is very unlikely. Based on this, we conclude that graphical languages are indeed less likely to be typed that textual languages. It would be interesting to see where in the spectrum languages with combined syntax fall, but unfortunately we do not have information on enough of those to draw any conclusions.

Given that a significant number of DSLs are typed, our next question is how well these type systems are described. In Figure 2.3, we show the fraction of typed DSLs that have a discussion🟩 of their type system in one of the papers that feature them. Because it makes little sense to expect type system discussions from untyped languages, they are not represented in this figure. Again, we assign DSLs to the year the first paper about them we found was published. Note that the category “example only” actually corresponds with the “low” quality assessment, as described in Section 2.2. The category “Discussion" actually corresponds quite well with the “high” assessment, because the number of languages that have a formal or informal description but no example, and would thus be assessed as “medium" is only 6, and thus negligible. For completeness, we also note that the number of DSLs assessed as “none” is larger, namely 20 instances, but still small compared to the total number of DSLs in the figure. We observe that the type system of most DSLs is not discussed in a paper, though the number of DSLs with type system descriptions is not negligible. The absolute number of languages is increasing, though we do not have enough data to support any conclusions concerning the evolution of type system descriptions in a relative sense.

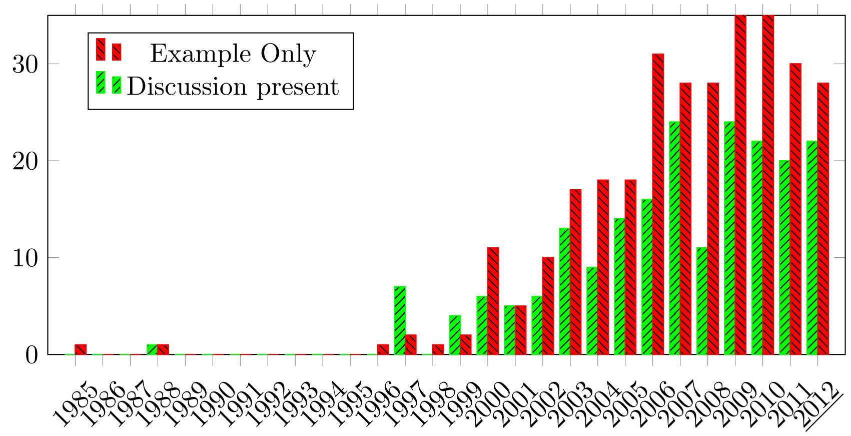


Figure 2.3: DSL Type System Discussion: Discussion present for a number of DSLs. N = 528.

	Language			Discussion	
Syntax	Typed	Untyped	Unknown	Present	Absent
Textual	745	684	100	308	1,221
Graphical	24	138	11	4	169
Combined	0	3	0	0	3

Table 2.3: DSL Type System related to Syntax: Graphical languages less likely to be typed.

A possible explanation for the lack of type system descriptions of DSLs is that size limitations common in conference papers prevent authors from including discussion of all aspects on a language. If that is a significant factor, we would expect that languages published in journal articles to have a type system descriptions more often. This relation is visualized in Table 2.4. In this table, we show the number of DSLs that do have type system descriptions, the number that do not and the percentage that do, split between DSLs with only conference papers, DSLs with only journal papers, and DSLs with both. Contrary to our expectation, the fraction of DSLs that have a type system discussion is not much different between DSLs published in conferences and journals. Note though that the number of “journal" languages we have is quite low, limiting the statistical significance of that result. Actually, if we apply a chi-square test to determine whether the presence of discussion is independent of source type, we get a p-value of 0.45, which means we cannot reject the hypothesis that the distributions are equal. The fraction is a little bit higher for DSLs that are described in both conference and journal papers, but not to a statistically significant extent.

Source	Present	Absent	Percentage
Both	26	33	44.07
Conference	263	396	39.91
Journal	24	44	35.29
Unknown	2	3	40

Table 2.4: DSL Type System Discussion per Source Type: Not Influenced By Paper Source.

Another explanation could be that a subsequent conference or journal paper on the same language should contain sufficient new material. If the type system was not covered in the first paper, it may be a way to justify the second. In order to test the hypothesis that type system discussion may be omitted due to space reasons, we also related the presence of type system discussion to the number of papers we have collected for each DSL. For this table, we only considered DSLs for which we are confident they are typed, which is true for 279 languages. If a language is untyped, or if we could not establish if it was typed or not, it was left out. In Table 2.5, we can see that the majority of DSLs in this study is represented by only one paper. More surprisingly, we observe no significant increase in the percentage of languages that have type system discussion as the number of papers increases to two. There is a small increase for DSLs with three or four papers, but the numbers of DSLs in these categories are too low to draw meaningful conclusions.

Given that type system discussions are relatively uncommon, the question arises whether it is worth looking for them. To see if the information we have on type systems with discussions is better than what we have for those without, we considered the attribute with the largest number of DSLs with classification

Number of Papers	No Discussion	Discussion	Percentage	Total
1	145	102	41.29	247
2	14	10	41.67	24
3	3	3	50	6
4	1	1	50	2

Table 2.5: DSL Type System Discussion by Number of Papers: Not Influenced By Number Of Papers.

"Unknown", overloading. In Table 2.6, we show our information on overloading split by the presence or absence of discussion. We can see that, while there still a lot of uncertainty, the percentage of unknowns drops from 80% to 50%, indicating that type system discussion, if present, does add useful information. In fact, if we apply a chi-squared test to see if the two groups are drawn from the same distribution, we find a p-value of less than 0.005, indicating it is very unlikely that the value we found for overloading is independent from the presence of a type system discussion.

	Overloading		
Discussion	Present	Absent	Unknown
False	41	11	271
True	63	19	121

Table 2.6: DSL Type System Discussion related to Overloading: Discussion Adds Information.

If the type system of a language is not described explicitly, we might still be able to use examples as sources of implicit information. In Figure 2.4, we show the fraction of DSLs that have examples in at least one of the papers discussing them. For some papers, it was unclear to if a given piece(s) of code were examples of the DSL, pseudocode or of a different language. The languages in these papers are classified as unknown in Figure 2.4. Similarly to Figure 2.3, we excluded the untyped languages here. Unsurprisingly, we observe that we have examples for the majority of DSLs. Though these examples can be limited in size, because they describe directly how a user might perceive the language, we still consider them valid sources of information.

As described in Section 2.2, in addition to whether DSLs are typed, we also collected information about what features type systems of typed DSLs have. As mentioned, we only collected this data for external, typed DSLs where we have found either a type system discussion or an example model. For many features, we unfortunately found that our data did not contain interesting patterns, or was too sparse to draw statistically valid results. One of the features where we did find interesting information was object orientation. In Figure 2.5, we show the number of DSLs featuring objects over the years. We observe again that most

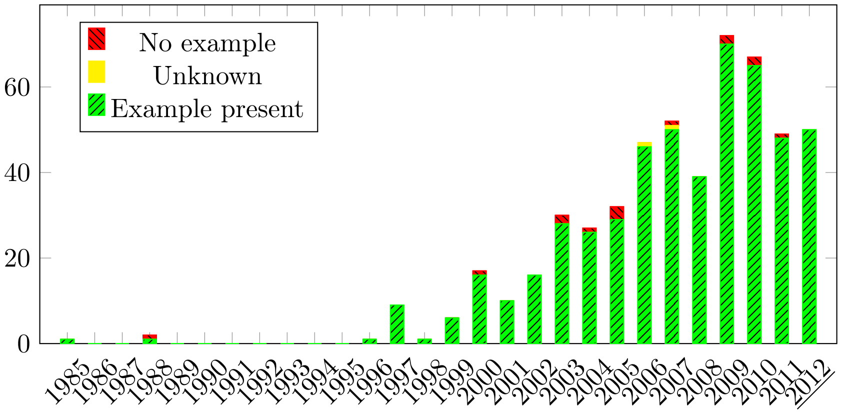


Figure 2.4: DSL Examples: Most Languages Described Using Examples. N = 528.

DSLs do not feature objects, but also that object-orientation appears not to get more popular over time, as we initially expected. While the absolute number of languages with objects increases, this increase is proportionate with the increase in the total number of languages. If we apply a Pearson Correlation test to the data, connecting the number of object-oriented languages to the total number of languages for each year, we get an R value of 0.97, which indicates a strong correlation, and a significance value of less than .0000001, indicating strong significance. The corresponding scatter plot is shown in Figure 2.6. The top plot shows the relation between total languages and object-oriented languages. The bottom plot shows the deviations from the expected relation.

Continuing our survey of DSL type systems, we look more closely at how the properties of DSL type systems evolved over time, and how they are related to each other. Overall, we found no significant shifts in the prevalence of various properties over the years. We also found no relation between the source of the papers used, and the level of detail of the DSL description. We did find that the number of typed, external DSLs continues to be significant, indicating it will remain a valid group of languages to target in the future.

Analysis

In addition to data on properties of DSL type systems, we also collected data on the types present in DSLs. However, due to the limited information available for many DSL type systems, we are limited in our ability to draw statistical conclusions based on this data, because we cannot be sure we extracted all types for all languages. Additionally, we have to be careful in comparing types

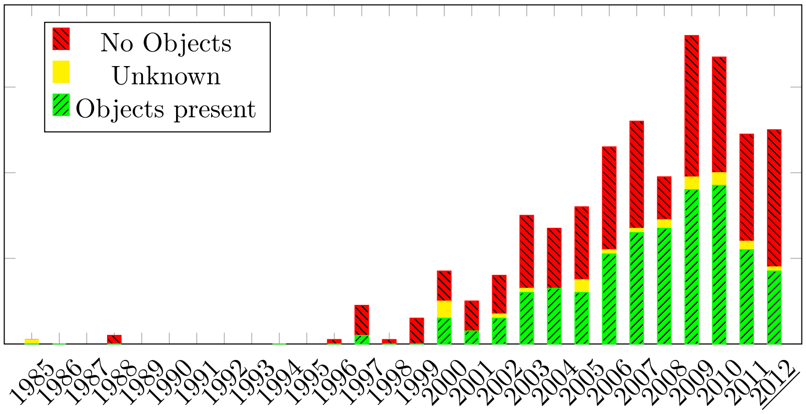


Figure 2.5: DSL with Objects: Most DSLs Do Not Feature Objects. N = 528.

between languages designed for different domains. For example, we could look at the number of types a language has a measure of its complexity. However, in one domain, it might make sense to have a wide range of types to represent specific kinds of numeric values, because the distinction between them is relevant. In other domains, it may make more sense not to have numeric values at all, and thus no numeric types. One conclusion we can draw, however, is that many DSLs use what could be called “common" types, like `integer` and `string`, that are also present in many GPLs, but surprisingly few, at a rough estimate one in five of our typed, external, documented languages, actually use domain-specific types. It would be interesting to see how strongly the presence of domain-specific types relates to the domain of the language, i.e. if some domains just do not require domain-specific types. Unfortunately, due to constraints on the time available compared to the time required, we choose not to attempt to classify the domain-specific languages we studied by domain, which means we cannot perform that analysis. Another explanation could be that modelers or language designers are so used to working with types from GPLs that they do not feel domain-specific types are needed. Again, due to limitations on the available time, we felt a more thorough investigation into the motivations of DSL designers fell outside the scope of this thesis. One observation that can be made is that the most common data types are numeric, and specifically integers. This suggests that numbers are considered useful in modeling in many domains.

In addition to looking at the properties of the DSLs and type systems described in the papers, we also looked at the influence of the venue where the DSL papers are published on their contents. The main results of this view are shown in Table 2.8. In order to create this table, we counted the number of DSLs published in each conference and journal. From the table, we can directly see that the Model Driven Engineering Languages and Systems (Models) conference is the most popular conference for publishing papers on domain specific languages, with a total of 102. The second most popular venue is the Soft-

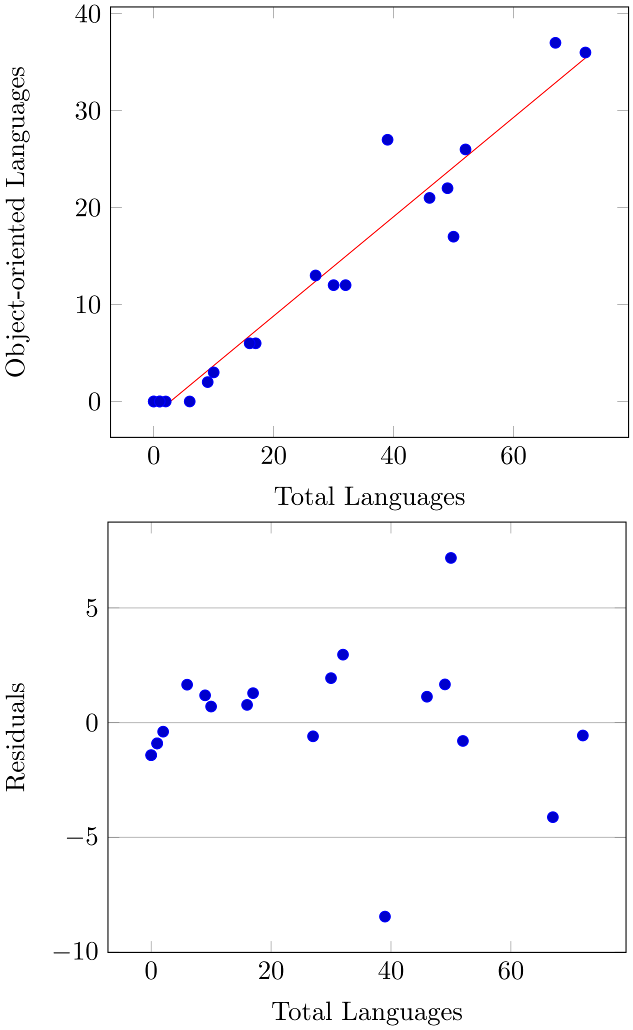


Figure 2.6: DSL with Objects: Scatter Plot confirms correlation between total number of languages and object-oriented languages.

ware & Systems Modeling (SSM) journal, followed by the European Conference on Modelling Foundations and Applications (ECMFA) conference, the Software Language Engineering (SLE) workshop and the ReasoningWeb journal. A problem with this comparison, of course, is that these conferences and journals are not all equal, or even similar, in size and age. In order to show the relation between the number of publications and the number of DSLs, we have collected data on the number of language papers published on average per year in the top 5 conferences in Table 2.8, namely Models, ECMFA[4], SLE, Generative Programming and Component Engineering (GPCE) and OOPSLA/SPLASH. The results are shown in Table 2.9. In this table, we show the conferences, with their names in abbreviated form for space reasons. Next, we first show the total number of language papers published in the conferences, then the number of editions in the studied period, and the average number of language papers per year we have data on. We assume that the sizes of the conferences are relatively stable, which means averaging the number of papers over the editions gives us a fair indication of their relative sizes. From among these, we can see one outlier in terms of languages published per year. SLE is a relatively new conference, but already has 65 language papers in its 5 editions. This confirms that, in contrast with the other, more general conferences, SLE really is a language-focused conference. We next show the percentage of the languages for each conference that have documentation, are typed, are external and are interesting according to our definition in Section 2.2 on 16. In these categories, we see little difference between the conferences. We do observe that languages published at ECMFA and GPCE are more likely to be documented and external, but the difference is only a little more than a standard deviation, meaning the results is not statistically very meaningful.

Threats to Validity

As for all literature reviews, there are several threats to the validity of the results of our SLR.

- Our search for DSLs was, due to time constraints, limited to DSLs described in papers published by three major scientific publishers, ACM, IEEE and Springer. Obviously, there are a significant number of DSLs that are not covered by this, for example because they were constructed by commercial companies who consider them trade secrets, because they were described in papers published elsewhere, in particular in non-computer science conferences and journals, or because they are described not in a paper, but in a book or a thesis. While we cannot eliminate this threat, we tried to minimize it by making sure that we covered as many DSLs as possible, from a variety of disciplines and publication dates. By doing

[4] Until 2009, this conference was called ECMDA-FA (European Conference on Model Driven Architecture®: Foundations and Applications), (ECMDA-FA). In this thesis, we will use the new name when referring to the conference in general.

Source	Languages
Model Driven Engineering Languages and Systems(Models)	102
Software & Systems Modelling(SSM)	79
European Conference on Modelling Foundations and Applications(ECMFA)	68
Software Language Engineering(SLE)	56
ReasoningWeb	45
Generative Programming and Component Engineering(GPCE)	45
OOPSLA[4]/SPLASH[5]	44
Practical Aspects of Declarative Languages(PADL)	41
International Conference on Model Transformation(ICMT)	39
Domain-Specific Languages(DSL)	38
Conference on Advanced Information Systems Engineering(CAiSE)	33
TOOLS	32
European Conference on Object-Oriented Programming(ECOOP)	31
Generative and Transformational Techniques in Software Engineering(GTTSE)	30
International Conference on Web Engineering(ICWE)	27
Transactions on Aspect-Oriented Software Development(TAOSD)	23
International Conference on Compiler Construction(CC)	23
Automated Software Engineering(ASE)	22
International Conference on Human-Computer Interaction(HCI)	21
Formal Methods for Components and Objects(FMCO)	19
Fundamental Approaches to Software Engineering(FASE)	19
International Conference on Software Reuse(ICSR)	18
International Conference on Software Engineering(ICSE)	18
International School on Formal Methods(SFM)	16
SDL Forum	16
International Conference on Service Oriented Computing(ICSOC)	16
Applications of Graph Transformations with Industrial Relevance(AGTIVE)	16
Implementation and Application of Functional Languages (IFL)	15
International Software Product Line Conference(SPLC)	14
Information Security Solutions Europe(ISSE)	14
International Federation for Information Processing(IFIP)	14
Hawaiian International Computer Science Symposium(HICSS)	14
International Conference on Conceptual Modeling(ER)	14
Algebraic Methodology and Software Technology(AMAST)	14
ADA-Europe	14

[4] Object-Oriented Programming, Systems, Languages & Applications

[5] Systems, Programming, Languages, and Applications: Software for Humanity

Table 2.8: DSL Publishing Venues: Models most popular.

this, we increase the chance that if not each individual DSL, at least all significant trends in DSL design are covered by at least one representative.

- In addition to the limits of our search area, we also have to consider the limitations of our search engine queries and the search engines themselves. Recall that for ACM and IEEE, we searched for "Domain specific language" as a keyword. Because this phrase is commonly accepted in the MDE community as a label for these languages, we expect that we did not exclude a significant number of papers by using this search criterion. While the choice of search phrase means that papers where the authors choose the shorthand "DSL" over the longer version are excluded, but in our experience, the shorthand, if it is included, is included in addition to the longer version, not instead of it. Keywords are required for all ACM and IEEE papers, so we also expect the search engine will be able to find all papers with this keyword without problems. In the case of Springer, because we could not search based on keywords alone, we instead used full-text search for the same phrase. Because the text of a paper includes any keywords, this will automatically include all papers that would have been found if we had been able to search based on keywords. We do assume that the search engine has access to the correct full text of all papers, but because all papers were published in the year 2000 or later, we feel we can assume this is the case.

- In addition to DSLs found in papers discovered using search engines, we also looked at languages described in [34]. This introduces a threat to validity, because the languages described in the paper are certainly not all the DSLs published before the year 2000. Still, because the authors of the bibliography chose the papers carefully, we feel that we still have a valid set of DSL representatives, that should allow us to observe any strong historical tendencies.

- Another significant threat to the validity of the results lies in the subjective judgement involved in the decision if a given DSL has a certain property or not. In part, this is caused by the diversity of the subject area, which makes it impossible to devise a uniform test that can be applied to all languages equally. We tried to minimize this risk by limiting ourselves to a restricted set of pre-defined concepts. By looking for the same characteristics for a concept in each language, we reduce the change for languages to be classified inconsistently.

- Another threat lies in differences between descriptions and reality. We look only at how DSLs are described in papers, not at actual tools, which means that we would not notice if a language is described as having an elaborate type system, which is not actually implemented. On the other hand, a language may have a type system that is not described in any paper. We tried to minimize this threat by paying special attention to examples of actual DSL models. If the description of the type system is correct, the features described should be present in the example(s) also. This still leaves the possibility that while the example contains types, the actual tools simple ignore them. However, because all languages presented as useable, we feel we can assume they do not contain superfluous elements.

Conclusions

The primary goal of this survey was to answer RQ 1: What features do DSL type systems have? Based on our survey, we conclude that most DSL type systems are static and strong, with more advanced features like object-orientation and type inference significantly less common. We observe that despite a marked increase in the number of languages, this has not changed over the period of the SLR. In our choice which formalisms would be suitable for type system definitions, these were the features we knew had to be supported. This will be discussed in greater detail in Chapters 3 and 4. We also selected the languages we used in the case studies in Chapter 5 based on the results of this survey. We also found that DSLs are published in a wide variety of venues, both language-oriented and not language-oriented.

2.3 Stakeholders

In addition to the place the type checker has in the model creation process and the properties of existing DSLs, it should also be useful to look at the type checker from the perspective of the people involved in its development and use. This is expressed in RQ 2. Usually, most of the people who come into contact with the type checker will be users of the language, who create their models using the DSL. Other stakeholders include the designers of the DSL and the implementors of the tool set for the DSL.

Language users

Language users are those that create models in the DSL. Their main interest is the type checker tool. Language users use the type checker to detect errors in their programs and models, and to prepare them for further processing. In order to be usable for this purpose, the type checker needs to be **correct** and **consistent**. If the type checker produces types that are incorrect or different from run to run, it becomes very hard for users to get the semantics they want. Additionally, the type checker must balance **power** with **efficiency**. A more powerful type checker can derive more information based on less input from the user. The more powerful a type checker is, the more computations it needs to get the desired result, which makes it less efficient. If the type checker becomes too inefficient, it needs several minutes for even small programs. This point can actually quite easy to reach, because the number of possible type combinations that might be valid for a given program can grow exponentially. If the type checker tests each possibility separately, this becomes computationally costly quite quickly. Finally, a good type checker should provide **useful feedback** about any errors it finds. If a type checker gives wrong or confusing error messages, the process of correcting the error is more difficult and time-consuming.

Language designers

Language designers are those that create the syntax and semantics of a language. They have to select what *constructs* the language has, how users can create instances of those constructs and what meaning these constructs have in the context of the complete program. In this thesis, we assume that language constructs have a textual representation, unless otherwise noted. In that case, the language designer needs to create a grammar to define how the parser should handle the model texts. The grammar serves to define the structure of the input for the scoping and type checking steps. Further on in the process, the output of the type checker is likely to be input for the implementation of the dynamic semantics. From the perspective of dynamic semantics, it is useful if all information relevant to a construct is easily accessible. In a tree structure, some information, like the relation between identifiers and their targets, is likely to be implicit, because the constraints that make the tree a tree prevent the information from being explicit. One solution is to represent the program or model as a more generic structure, for example an *abstract semantic graph* [35], instead of a tree. The removal of the extra requirements placed on trees over graphs creates room for the type checker to make implicit information explicit. In order to add information to a model, a language designer may want to **add elements to the model**, to **change the values** contained in an element, to **link two elements** or **change the structure** of an element. Because not every element can be handled in the same way, there also needs to be some way to **control** how these changes are applied. If the input program or model is inconsistent, it may actually be impossible to create an output model that satisfies all requirements. The language designer may want to define **error messages** that are triggered when an inconsistent situation is detected. Additionally, during the design process and as the language develops, language designers will likely consider various options for language constructs and their semantics. Thus, type system specification formalisms should allow **rapid implementation**, to make prototyping practical.

Implementation developers

Once there is a sufficiently complete design of a language, language implementors can start developing an implementation of the language. Note that this does not mean that the design is final, because insights derived from the implementation can affect the design of the language, resulting in language evolution [4]. In practice, language implementors will not develop every tool and component from scratch, but use either generators that create specialized components based on specifications, or generic components that can be configured to fulfill the required task. For our type system specification formalism, this implies that implementors would like to have an **interpreter** or a **generator** that can be used to implement the specification easily. Additionally, the resulting component needs to connect to other parts of the language tool set, so it should have a **flexible interface** that can communicate with a variety of other components.

2.4 Requirements

From our description of the domain and the priorities of the stakeholders, we compiled the following list of requirements:

Requirement 1: Type checking The specification language should allow the designer to specify what elements are typed correctly based on conditions. This relates primarily to the language designers, the primary users of the specification formalism.

Requirement 2: Type computation The specification language should allow designers to create new model elements based on model content. These elements can be used to store computed type information in the model, allowing it to be reused later. This again relates primarily to the language designers, the primary users of the specification formalism.

Requirement 3: Element transformation The specification language should allow elements of the model to be changed, updated and transformed based on computed type information. This also relates primarily to the language designers, the primary users of the specification formalism.

Requirement 4: Prioritization The specification language should support disambiguation by allowing designers to indicate which types should be preferred over which others. This is on the one hand related to language users, who want the type checker to accept as many valid models as possible, and one the other hand related to designers and implementors, who want to avoid ambiguities to simplify processing.

Requirement 5: Termination Application of a type system specified in the specification language to a model should always terminate. This relates primarily to language users, because they want to be sure that the type checker will process their models in a reasonable amount of time.

Requirement 6: Flexible input and output The type system definition should be useable for a variety of different parsers, template engines and model transformation tools. This primarily relates to engineers that implement the language, who want to have an unrestricted choice in how they design the other components of the language tool set.

Requirement 7: Implementation The type system specification should be easily or even automatically implementable, and the resulting type checker should be correct and consistent. This relates to both language users, who use the type checker the most and language implementors, who are responsible for creating and maintaining it.

2.5 Conclusions

In this chapter, we have studied the properties of domain specific languages, and looked at the properties that a type system language would need to be able to describe their type systems formally and effectively. We did this through a systematic literature review, where we looked at papers published on DSLs in journals and conference proceedings published by several major publishers, and through an analysis of the stakeholders of DSLs and their priorities and requirements. Based on the combined results of the SLR and primarily the stakeholder analysis, we formulated a number of requirements in Section 2.4. In short, we want a type checker to be correct, efficient and consistent, and the type system language should enable that. In the longer term, the formalism should support the development of the DSL by allowing easy evolution of the type system. In the next chapters of this thesis, we will look at actual formalisms and to what extend they meet these requirements.

Conference	Papers	Editions	Average	Languages	Documentation	Typed	External	Interesting
Models	116	12	9.67	120	89%	49%	97%	43%
ECMFA	78	8	9.75	80	98%	54%	99%	53%
SLE	65	5	13	67	87%	55%	93%	43%
GPCE	51	12	4.64	51	96%	63%	86%	51%
OOPSLA	47	13	3.62	47	85%	53%	83%	36%
Average					91%	54.8%	91.6%	45.2%
Std. Deviation					5%	5%	6%	6%

Table 2.9: DSLs related to number of papers: SLE stands out in total number of languages, but other properties mostly similar. “Interesting” language are typed and have either an example or a type system discussion in a paper.

Chapter 3

MSOS as Type System Definition Language

Evolution of programming languages requires co-evolution of static analysis tools designed for these languages. Traditional approaches to static analysis, e.g., those based on Structural Operational Semantics (SOS), assume, however, that the syntax and the semantics of the programming language under consideration are fixed. Language evolution is, therefore, likely to cause redevelopment of the analysis techniques and tools. Moreover, the redevelopment cost can discourage the language engineers from improving the language design. To address the co-evolution problem we suggest to base static analyses on modular structural operational semantics (MSOS). By using an intrinsically modular formalism, type rules can be added, removed or modified easily. We illustrate our approach by developing an MSOS-based type analysis technique for Chi, a domain specific language for hybrid systems engineering.

3.1 Introduction

Development of a programming language is an ongoing process: new language constructs are being introduced, superfluous ones are being removed and semantics of the existing ones is being reconsidered [4]. Traditionally, static analyses are, however, being developed under the assumption that the syntax and the semantics of the programming language under consideration are fixed. Traditional semantic specification methods such as Structural Operational Semantics (SOS) encourage this, due to the extensive changes caused by simple additions. Language modification is, therefore, likely to cause redevelopment of the analysis techniques and tools.

As observed in Section 2.3, the semantic analyses' based on the changing semantics should, hence, *co-evolve* together with the language syntax and semantics [65]. Moreover, static semantic analyses should *support decision making* by the language designers by providing them with insights on implications of the decisions taken on the analyses precision and complexity. To this end different design alternatives considered by the language designers should rapidly propagate to semantic analyses. To support both the language/analyses co-evolution and the decision making, we need a flexible formalism for specifying static semantics. Thus, the two research questions that are addressed in this Chapter are RQ 3 and RQ 4.

It has been suggested by the designers of the modular structured operational semantics (MSOS) [84,85] that it can be successfully applied to this end. In this chapter we validate this claim by implementing the form of static analysis that is the primary subject of this thesis, type analysis, for a challenging domain-specific language under development (Chi, evolving from Chi 1.0 to Chi 2.0), and verifying that

(i) the analysis developed is indeed flexible enough to co-evolve with the changing language, and

(ii) the prototype type checker required by the language designers can be implemented with a limited development effort, i.e., our type checker can support the decision making by the designers.

The programming language we consider in this chapter is Chi [10, 57], a specification language for hybrid systems. From the language perspective, Chi combines features of traditional procedural programming languages (e.g., loops, assignments) with domain-specific elements pertaining to continuity and parallelism. In terms of our survey, Chi is a typed, textual language. The Chi type system is both static and strong, as are most DSLs we found in our survey in Section 2.2. In terms of more advanced features, Chi is not object oriented and does not feature type inference, but it does have overloading and type parameters. This allows us to look how MSOS handles more complex type system concepts.

Chi 2.0 Example

To illustrate Chi 2.0[1] consider the following fragment:

```
1        proc ContrivedBuffer⟨T : type⟩(chan a?, b! : T) =
2        ⟦          cont     s: real
3                   var      x : T
4                   ,        xs : [T] = []
5        ::         eqn      s = SIN(time) + 3
6              ||
7                   *(       LEN(xs) < s ⟶a?x ;
8                                              xs := xs ++ [x]
9                   |        LEN(xs) > 0 ⟶b!HD(xs) ;
10                                             xs := TL(xs)
11                  )
12       ⟧
```

The fragment above illustrates a Chi 2.0 process definition. Structural language elements, e.g., repetition, parallel, alternative or sequential composition are typeset in the bold face, e.g., ***(. . .)**, ||, | and **;**, respectively. The built-in functions are typeset in small capitals: LEN, HD(head), TL(tail) and SIN.

Processes are used in Chi 2.0 to group behavior, in such a way that it can be instantiated and reused multiple times. Process ContrivedBuffer specifies the behavior of a buffer that receives values of some type T through channel a and sends them on through channel b. Angular brackets in the process definition indicate its polymorphic nature: ContrivedBuffer can be instantiated to manipulate data of any type T.

In lines 2–4 local variables are declared. The maximum size of the buffer is controlled by the continuous variable s, varying over time. The list variable xs stores the buffer values and is initially empty, and x stores temporarily a value just received or to be sent. Lines 5–11 describe the actual behavior of the process. The behavior consists of two parallel parts, combined by ||. The equation in line 5 determines the value of the continuous variable s as a function of a global variable *time*. The read-only variable *time* is part of the semantics of Chi 2.0. It is updated by the simulation environment and provides access to the current simulation time. The equation should be satisfied at all times during the process execution.

The lines 7–11 specify the discrete behavior of the process. The behavior consists of the repeated choice between two conditional alternative steps: receive (denoted **?**) the value x followed by appending x to xs, and send (denoted **!**) HD(xs) followed by removing the value HD(xs) from xs. If both alternatives can be carried out, the process non-deterministically chooses between them.

Chi specifications are usually being developed by mechanical engineers, often with limited training in formal methods or software development. As such, the Chi language designers aim at providing the specification developers with

[1] For the sake of simplicity we opt for the Chi 2.0 notation in the examples.

as much feedback on potential errors as possible early in the development life cycle. Chi employs prescriptive typing, i.e., types are provided by the developer while the type checker verifies whether the types are consistently used. For instance, expression $2 +$ "Hello!" should be rejected by a Chi type checker since addition cannot be applied to a number and a string. Expression $2 + x$ should be associated, however, with **int** as long as x itself is associated with **int**.

Chi 2.0 Function Call Example

Consider the following Chi 2.0 expression: $\textsc{take}([y > z, \mathit{true}] + +xs, \mathit{size} + 1)$. Here, the TAKE function takes a list, say l, and a natural number, say n, as parameters and returns a list of the first n elements of l, or l if $\textsc{len}(l)$ is less or equal than n. For this expression to be valid in Chi 2.0, the type checker should derive a type for it. In this example, we discuss how this would be using a top-down approach.

The top operation of this expression is the invocation of the TAKE function. The type of the result of TAKE depends on the first parameter, so TAKE has an implicit template that specifies this relation. In this example, the value for the list parameter is the result of the concatenation of a list of two booleans, one the result of a comparison and one a constant, and the variable xs. In Chi 2.0, all elements of a list must be of the same type, hence xs must be a list of booleans.

The value of the second, numerical, parameter is the result of the addition of the variable *size* and the constant 1. However, to "fit" as parameter of TAKE the result must be natural. This means that the type of *size* must be natural, because any addition involving integers or reals cannot have a natural result type in Chi 2.0.

Based on the types that the typechecker has derived for the parameters, it has to decide if this invocation of TAKE is valid and if so, determine what the result type is. The function needs a list and a natural number, and if the requirements mentioned above are met, the invocation is correct. The TAKE function has, as mentioned before, an implicit template. More specifically, the return type is a list with elements of the same type as the list provided as parameter. In this case, that means that the result type of the expression is list of booleans.

The remainder of the chapter is organized as follows. After introducing MSOS and Chi in Section 3.2, we review the evolution of typing schemes in Chi (Section 3.4) and discuss the formalization of type checking rules in MSOS in Section 3.5. In particular, we look at the two main requirements defined in Section 3.1. To address **(i)**, while discussing each language construct we stress the similarities and the differences of the formalizations corresponding to different typing schemes implemented or considered during different phases of the evolution of Chi. To address **(ii)** we have implemented the approach in a first prototype using Maude [29]. We discuss the implementation and lessons learned in Section 3.6. Based on our experiences, we developed a second prototype using ASF+SDF, described in Section 3.7. After reviewing the related work in Section 3.8 we summarize our contributions and sketch future research directions in Section 3.9.

3.2 Preliminaries

MSOS

In this section we briefly present MSOS, modular structural operational semantics, introduced by Peter Mosses in [84,85] to address the modularity shortcomings of SOS [90]. The modularity shortcomings of SOS are related mostly to difficulty of modifying the type system environment information: adding a new component, e.g., a stack, to the type system environment requires modification of all SOS rules. MSOS resolves this by making the structure of the environments implicit. Presenting the formal semantics of MSOS we follow [17,18].

> (cf. [17]) A specification in modular structural operational semantics (MSOS) is a structure of the following form. $\mathcal{M} = \langle \Omega, \mathit{Lc}, \mathit{Tr} \rangle$ where Ω is the signature, *Lc* is a label category declaration and *Tr* is the set of transition rules.

The *signature* defines function symbols and constants used in the language being specified. *Transition rules* are $\frac{C}{c}$ intuitively read "whenever all the conditions in C hold so does the conclusion c", where each condition in C is either a transition, a predicate or a label expression, and the conclusion c is a transition. Rules with the empty set of premises C are called *simple rules*: conclusions of simple rules should always hold.

Transitions are triples that can be interpreted as steps or rewritings. For example, $e_1 \text{=}\{\alpha\}\text{⇒} e_2$ is read as: configuration e_1 can make a step with the label α, or can be rewritten, to configuration e_2. *Label transformers* are ways to modify the labels, while labels in MSOS are morphisms associated with a given category. A *category* is a mathematical construction consisting of a class of objects O; a class of morphisms A between the objects, such that each morphism $a \in A$ has a unique source object $\mathit{pre}(a)$ and target object $\mathit{post}(a)$ in O; an associative partial composition function $\circ$ on the pairs of morphisms; and a function $\mathbb{1}$ mapping objects to identity morphisms, such that for any $a \in A$, $\mathbb{1}_{\mathit{pre}(a)} \circ a = a = a \circ \mathbb{1}_{\mathit{post}(a)}$. Formal syntax of the transition rules and label transformers can be found in [17].

Example 1. Consider the following rules from [17].

- The rule $\dfrac{n = n_1 + n_2}{n_1 + n_2 \text{=}\{\imath\}\text{⇒} n}$ says that if n is equal to the sum of n_1 and n_2, then the term $n_1 + n_2$ can be rewritten to n. In this case, the condition is a predicate and the conclusion is a transition, labeled with a special silent or unobservable label $\imath$. The label $\imath$ satisfies $\imath \in \mathbb{1}$, where $\mathbb{1}$ is the trivial category consisting of just one object and one morphism.

- The rule $\dfrac{e_1 =\{\alpha\}\Rightarrow e_1'}{e_1 + e_2 =\{\alpha\}\Rightarrow e_1' + e_2}$ reads "whenever e_1 can be rewritten to e_1' with respect to α, the summation term $e_1 + e_2$ can be rewritten to $e_1' + e_2$ with respect to the same label. Both the condition and the conclusion are transitions.
- Finally, $\dfrac{\alpha' = \mathbf{set}(\alpha, \mathit{env}, \mathbf{get}(\alpha, \mathit{env})[x \to v]) \qquad e =\{\alpha'\}\Rightarrow e'}{\lambda x(e)v =\{\alpha\}\Rightarrow e'}$ allows us to introduce x. The first condition is a label expression based on two "built-in" functions **set** and **get** that add and retrieve values from the label respectively. Label α' is, therefore, obtained from α by updating the value of *env* to include the mapping of the variable x to the type v. In general, $\mathbf{get}(l, \mathit{var})$ retrieves the value of the variable var in the context expressed by the label l, and $\mathbf{set}(l, \mathit{var}, \mathit{val})$ returns a new label identical to l except for the value of var being set to val. The second condition and the conclusion are transitions, that represent typing decisions. The second condition holds if the expression e can be rewritten to typed expression e' with label a'. If both conditions hold, the conclusion $\lambda x(e)v =\{\alpha\}\Rightarrow e'$, which states that the expression $\lambda x(e)$ can be rewritten to e' in environment α, also holds.

In the rules above, the label category *Lc* is $\mathbf{ContextInfo}(\mathit{env}, \mathit{Environment})(\mathbb{1})$, where $\mathbf{ContextInfo}(\mathit{env}, \mathit{Environment})$ is a label transformer derived from a set of environments *Environment*. Hence, e.g., $\mathbf{get}(\alpha, \mathit{env})$ denotes the environment corresponding to the label α.

Semantics of MSOS is given by means of a mapping to an arrow labeled transition system. Details of the mapping can be found in [17]. In particular, predicates are assumed to be available for use, and not part of the MSOS rules.

Hybrid Specification Language Chi 2.0

In this section we briefly present the syntax of Chi 2.0 [10]. For the sake of brevity and without loss of generality we slightly simplified the syntax. A Chi 2.0 *model* is composed of comma separated series of variable, channel and action declarations followed by **::** followed by a process description. Variables can be discrete (x and xs in Example 3.1), i.e., their values can change only by means of explicit assignments; continuous (s in Example 3.1), i.e., their values are determined by a continuous function of time; and algebraic, i.e., their values may change according to a discontinuous function of time. The change of continuous and algebraic variables can be restricted by a special form of process, called *equation*, e.g., **eqn** $s = \textsc{sin}(\mathit{time}) + 3$. Channels serve for unbuffered, synchronous communication between the processes that can send (**!**) data to channels or receive (**?**) data from them. Both variables and channels are *typed*. The atomic types are **bool**, **nat**, **int**, **real**, **string** and **enum** (enumerations). Type constructors operate on existing types to create structured types such as sets, lists, arrays, record tuples, dictionaries, functions, and distributions (for stochastic models). The type corresponding to a channel is the type of data that is communicated via the channel.

Processes can in their turn be atomic or composite. Atomic processes include performing an action, sending or receiving data, assigning values to variables, delaying or performing a blocking conditional action: if the condition does not hold the process will block until the condition becomes true. Furthermore, equations are considered as atomic processes. As seen in Example 3.1 composite processes can be obtained from the atomic ones by repetitive application of parallel, sequential or alternative composition, or by means of a loop. Furthermore, similar to models processes can be defined separately, i.e., processes also act as subroutines in Chi 2.0. As such the processes can be *parameterized* by a sequence of parameters (see line 1 in Example 3.1). Processes can be made even more flexible by the use of *templates* (e.g., $\langle T : type\rangle$ in Example 3.1).

As we are going to focus on type analysis, we are also interested in built-in functions, which describe mathematical computations without side effects, and operators. These include traditional operators on the booleans, numbers and strings, explicit type conversions, and constructors for composite types, like lists and sets. Similarly to processes functions can make use of templates. *Explicit* templates require the user to supply values for the template parameters, which are then used to instantiate the function. Similarly to Example 3.1 explicit template parameters are listed between angular brackets: $\mathbf{func}\ f\langle T : \mathbf{type}\rangle(\mathbf{val}\ a : T) \to T$. A common use for function templates is to allow a function to be applied to multiple types of parameters: $f\langle \mathrm{nat}\rangle(7)$. It should be noted that we write $f\langle \mathrm{nat}\rangle(7)$ rather than $f\langle \mathbf{nat}\rangle(7)$ to distinguish between strings representing types in the Chi 2.0 program ("nat"), referred to as type literals, and actual types (**nat**).

In addition to explicit templates, Chi 2.0 supports functions with implicit templates. *Implicit* templates are instantiated during function invocation. In contrast to explicit templates, the values for the implicit template parameters are determined by the type checker, based on the types of the function parameters supplied by the user. Implicit templates are mainly used to allow functions like $\textsc{hd}()$ to work for lists with any type of element, while still allowing the type checker to derive a useful return type.

Example 2. In the following function definitions we list the implicit type parameters between [and].

The function $\textsc{len}()$ calculates the length of a list: $\mathbf{func}\ \textsc{len}[T : \mathbf{type}](\mathbf{val}\, xs : [T]) \to \mathbf{nat}$. The type of the elements of the list does not influence the return type, which is reflected in the function declaration by the fact that T is only used once.

Next consider $\textsc{hd}()$ defined as $\mathbf{func}\ \textsc{hd}[T : \mathbf{type}](\mathbf{val}\, xs : [T]) \to T$. Observe that T occurs both in the parameter type and as the return type. The function $\textsc{hd}()$ returns the first element of a non-empty list, so here the actual type of the elements of the list does affect the type checking process.

Finally, the definition of $\textsc{sort}$ uses T to describe types of multiple parameters $\mathbf{func}\ \textsc{sort}[T : \mathbf{type}](\mathbf{val}\, xs : [T], f : (T, T) \to \mathbf{bool}) \to [T]$. The first parameter of function SORT is a list of type T, and the second one is a predicate $f : (T, T) \to \textbf{bool}$ imposing a sorting order on the elements of type T. □

Functions are first-class citizens in Chi 2.0: a function can, e.g., be passed as a parameter to another expression (function). One construct using functions as parameters is a *fold*. Folds have four parameters: $\langle f, i \leftarrow I, p(i), c(i)\rangle$. I is the iterator generating values and storing one value at a time in i. Once a value is generated, predicate p is applied to it, and if $p(i)$ is *true*, then the auxiliary value $c(i)$ is computed. Finally, the aggregation function f is applied to all the auxiliary values computed. Chi 2.0 allows only a restricted number of aggregation functions such as $+$, $*$, MAX, to be used in folds.

Example 3. Let $\langle +, i \leftarrow [1, 2, 3, 4], i > 2, i*i\rangle$ be a fold. Then, each value from the list $[1, 2, 3, 4]$ is in its turn assigned to i, compared with 2, and for those values larger than 2 the square is computed. Finally, all squares are added. Hence, the value of $\langle +, i \leftarrow [1, 2, 3, 4], i > 2, i * i\rangle$ is $3 * 3 + 4 * 4$, i.e., 25. □

3.3 MSOS and MSDF: Specifying a Type Checker

Our approach starts from a formal specification of a type checker in MSOS. To simplify the generation process, we choose to represent the MSOS in a textual format using a variation of the MSDF [27].

In addition to MSOS rules, MSDF also supports declarations, formulas and modules. *Declarations* describe algebraic data types and variables of those types. In our case, algebraic data types are used to represent the structure of the input tree, the type values and (possibly) the output tree. *Formulas* can then be used to give initial values to the variables or link them to each other. In a type system definition, formulas can be used to initialize the environment, e.g., by introducing default functions. In our Chi 2.0 specification, we have not used formulas yet, thus we have decided to omit them from the example. Finally, *modules* increase readability of the semantic rules by allowing the related rules to be stored together. Modules can be imported as a whole into other modules.

Example 4. Consider the MSDF specification with rules for the type system presented in Figure 3.1. This module consists of three main parts, *imports*, in Line 1, *declarations*, in Lines 3–8 and *transition rules* in Lines 10–33. The imports are part of the modular structure of MSDF, and allow transition rules and declarations from other modules to be used in this module. While the example module could, and in practice should, be split into multiple modules, here it was not done for the sake of presentation. We import generic definitions for identifiers `Id` and basic data types `Bool`, `Int` and `Real`. These modules also define variables for these types: variables with names consisting of `ID` (`B`, `I`, `R`) followed by zero or more digits are considered to be of type `Id` (`Bool`, `Int`, `Real`, respectively). Note that the data type definition method in MSDF is not suited for efficient definitions of the integers and the reals, so `Int` and `Real` are actually defined outside MSDF.

```
 1 see Bool, Id, Int, Real.
 2
 3 E:Exp  ::= sumfold(Exp,Exp,Exp) | less(Exp,Exp) |
 4            dot(Id) | Id | Int | Real.
 5 T:Type ::= bool | int | real | func(Type*,Type).
 6 D:DType :: cont | disc.
 7 SENV : StatEnv = (Id,Type)Map.
 8 DENV : DynEnv = (Id,DType)Map.
 9
10 I:Exp ===> int.
11
12 R:Exp ===> real.
13
14 lookup(ID,SENV) = T
15 ---------------------------
16 ID:Exp =={statenv=SENV,---}=> T .
17
18 lookup(ID,SENV) = real, lookup(ID,DENV) = cont
19 ---------------------------
20 dot(ID):Exp =={statenv=SENV, dynenv=DENV,---}=>
     real.
21
22 E1 =={---}=> T1, E2 =={---}=> T2,
23 max(T1,T2) = T3, max(T3,real) = real
24 --------------------------
25 less(E1,E2) : Exp =={---}=> bool .
26
27 Exp1 =={---}=> T1, iscontainer(T1) = true,
28 element(T1) = T2
29 E2 =={---}=> func(T3,bool), max(T2,T3) = T3,
30 E3 =={---}=> func(T4,T5), max(T2,T4) = T4,
31 max(T4,real) = real
32 --------------------------
33 sumfold(E1,E2,E3) =={---}=> T4 .
```

Figure 3.1: MSDF specification of a type checker for a fragment of Chi.

Lines 3–6 contain declarations for the types `Exp`, `Type` and `DType` and variables of those types. The first type defines possible expression constructs, in the form of a list of options separated by the "`|`" symbols. An expression in the listing above is, therefore, `sumfold` applied to three expressions, representing a fold, as described in Section 3.5, `less` applied to two expressions, a boolean predicate comparing two values, `dot` applied to an identifier, an unary operator that computes a derivative, an identifier, an integer or a real number. "`E:Exp`" means that variables with names consisting of `E` followed by zero or more digits are considered to be of type `Exp`. In the next line, the type `Type` defines three constants representing the basic types: `bool` for boolean values, `int` for integer values, `real` for real values, and `func(Type*,Type)` for functions from lists of values of some type to a single value. As above, variables with names starting with `T` and followed by one or more digits are of the type `Type`. We stress that while `Bool`, `Int` and `Real` are sets of values, `bool`, `int` and `real` are constants representing the types of these values. In the last line, the type `DType` defines two constants describing continuous and discrete behavior respectively.

Lines 7–8 contain declarations for the types "`SENV`" and "`DENV`" and variables of those types. These declarations refer to the predefined "`Map`" type, and are used when referring to components from the environment.

Lines 10–33 contain the transition rules. The first two (Lines 10 and 12 are simple rules, without explicit preconditions. Implicitly, the types of the variables used restrict their values to integer and real constants, respectively. Because these rules are not affected by the environment, we do not mention it. Implicitly, this means the rule does not change the environment. In the case of the rules at Lines 14–16 and 18–20, the preconditions are based on the built-in `lookup` function. The `lookup` function is used to inspect the environment components, such as `statenv` and `dynenv`. Given a list of key-value pairs and a key that occurs in the one of the pairs, this function returns the corresponding value. If the key does not occur in the list of pairs, a special value "none" is returned. The rule in Lines 14–16 reads, thus, "If `ID` is known to have type `T` in the static type component of the environment, any reference to `ID` also has type `T`"; and the rule in Lines 18–20 "If `ID` is known to be a continuous real variable, then application of the `dot` operator results in a value of type `real`". These rules both reference the environment. They gain access to the relevant information by extracting components from the label. If there are other components present in the environment which are inspected or changed by the rule, they remain unchanged, as indicated by the "`--`" marks.

The next transition rule (Lines 22–25) has four preconditions. The first precondition requires `E1` to be of a valid type `T1`, i.e., transition `E1 =={--}=> T1` to be possible. Similarly, the second precondition requires `E2` to be of a valid type `T2`. The remaining preconditions use an auxiliary `max` function. Based on a partial order on types, `max` returns the larger of two types, if they are comparable. In the example we assume `int` $\prec$ `real`. If the `max` function returns a type `T3` and that type is `real` or below it in the type order, we conclude that `T1` and `T2` are numerical types, and hence, `E1` and `E2` can be compared with the `less` operator.

The last transition rule (Lines 27–33) is an example of an even more complex rule, in this case of a fold based on the addition operator. When a fold expression is evaluated, it iterates over the collection represented by `E1`. The function represented by `E2` is then applied to each value, and acts as a filter: if the result is "`false`" the value is discarded. Next, the function represented by `E3` is applied to each value that passes the filter. The results of this function are combined into one value using the "`addition`" function, which is the result of the fold.

When we look at folds from a typing perspective, this means that `E1` needs to be of a collection type, so the fold can take elements from it. The "`collection`" predicate tests for this, while the "`element`" function retrieves the type of the elements of the collection. The next two components must have a function type that can accept the collection elements as parameters. Additionally, the function serving as a filter must return a boolean value, and the function that creates values to be combined must returns values that can be combined by, in this case, the "`addition`" operator. The type of the fold expression itself is the same as that type, as long as it is numeric.

3.4 Evolution of numerical types in Chi

In order to illustrate how the type checker expressed in MSOS can co-evolve with the language itself we focus on a part of Chi static semantics that significantly evolved between versions: the type system for numerical values. Chi has three types of numerical values: **nat** for natural numbers, **int** for integer numbers and **real** for real numbers. Mathematically, there is an obvious relation $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{R}$. This, however, may or may not be reflected in the type system.

Earlier versions of Chi, such as Chi 1.0 [57][2], insisted on a strict separation of the numerical types, i.e., not performing the type widening at all. This approach was motivated by the fact that for the specification engineers the natural numbers usually represent quantities, while the real numbers correspond to measurable aspects of physical artifacts such as speed, pressure, or frequency. Moreover, Chi is also used for training system engineering students with no previous programming experience, and this audience should be made aware of differences between mathematical sets such as $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ and programming language types **nat**, **int** and **real**.

Strict separation of types in Chi 1.0 implied, unfortunately, that $\textsc{sin}(0)$ cannot be computed as the sine is defined as a **real** → **real** function. Therefore, later on the language designers considered applying full widening on numerical types, i.e., assuming $\mathbf{nat} \prec \mathbf{int} \prec \mathbf{real}$. While Chi 1.0 would disallow $1 + 2.5$ as addition is being applied to arguments of two different numerical types, this version, that we call "Chi 1.5"[3], would convert 1 to 1.0 and calculate the result. In this way, however, the distinction between different numerical types becomes blurred. From a computer science perspective, this is not really significant, but from a hybrid systems viewpoint, different numerical types are used for physically different things, which should not be confused. Therefore, while developing Chi 2.0 [58] the language designers have chosen to apply type widening solely to constants explicitly mentioned in the model. In this way the designers felt that novices will still understand the distinction between mathematical sets and types in Chi 2.0, without being forced to write 0.0 instead of 0 just to pacify the

[2] Though the report was finalized in 2008, initial steps in the development of Chi 1.0 were made in 1995 and the language was in active use from 2005.

[3] "Chi 1.5" is a name we use for the sake of convenience rather than an officially released version of Chi.

type checker. In order to facilitate this, we created two new types, **cnat** and **cint**, for constant numeric values, to allow a separate treatment. This solution has been reported by the language designers as preferred to both Chi 1.0 and Chi 1.5.

3.5 Typechecking Chi in MSOS

As explained in Section 3.2, MSOS rules describe transitions between configurations of transition systems. A computation, then, is a sequence of transitions from an initial to a final configuration. In the case where a type system has been specified using MSOS, the initial configuration is a program expression to be typechecked, and the desired final configuration is the type corresponding to that expression. In this section, we will describe our definition of the Chi 2.0 type system in MSOS.

While specifying the MSOS transition rules we opt for *big-step semantics*, i.e., we describe how the overall results of the executions are obtained as opposed to the *small-step semantics*, describing how individual steps take place. As recognized by Mosses in [85] big-step semantics is preferable for specifying type-checking. As static semantics in this case involves environments not changed by the type-checking rules in a corresponding MSOS, most labels are identity morphisms. We omit those labels for the sake of readability.

In the remainder of this section we present MSOS rules for three typing schemes considered in Section 3.4: no widening (Chi 1.0), full widening ("Chi 1.5") and widening restricted to constants (Chi 2.0). We stress how the rules can express changes in the typing schemes.

Basic type rules

The basis of every type system lies in the atomic expressions, such as literal constants and variable references. To derive types for constants we use simple rules as typing of these constants does not depend on any precondition. Labels in small capitals are used for the ease of reference only and should not be considered as a part of the MSOS syntax:

$$\frac{}{\mathit{true} ==> \mathbf{bool}}\ (\textsc{True}).$$

Unfortunately, this approach would require providing a separate transition rule for *each* constant. As this is not feasible for numerical types we introduce unary predicates *natural_number*, *integer_number* and *real_number* that evaluate to *true* according to the syntax described in [57]. Formally, *natural_number* evaluates to *true* if its argument is a digit zero, or a non-empty sequence of digits starting with a non-zero digit; *integer_number* evaluates to *true* if its first character is $+$ or $-$, and the remainder is either a digit zero, or a non-empty sequence of digits starting with a non-zero digit. Finally, *real_number* evaluates to *true* if its argument is composed of two parts divided by a point, and each one of the parts is either a digit zero, or a non-empty sequence of digits starting with a non-zero digit.

Using these predicates for Chi 1.0 and Chi 1.5 we write

$$\frac{\mathit{natural\_number}(n)}{n ==> \mathbf{nat}}\ (\textsc{Nat}).\quad \frac{\mathit{integer\_number}(z)}{z ==> \mathbf{int}}\ (\textsc{Int}).\quad \frac{\mathit{real\_number}(r)}{r ==> \mathbf{real}}\ (\textsc{Real}).$$

The typing scheme of Chi 2.0, however, requires refining the type system to include special types for natural (**cnat**) and integer (**cint**) constants, and updating the transition rules as follows:

$$\frac{\mathit{natural\_number}(n)}{n ==> \mathbf{cnat}}\ (\textsc{Nat}).\quad \frac{\mathit{integer\_number}(z)}{z ==> \mathbf{cint}}\ (\textsc{Int}).$$

In order to determine the type of a variable one should consult the environment. Similarly to Example 1 we use the built-in function **get** to derive the current environment. Since the environment is a morphism it can be applied to the variable name to obtain the variable's type:

$$\frac{\mathbf{get}(\alpha, \mathit{env})(x) = v}{x = \{\alpha\} \Rightarrow v}\ (\textsc{Lookup}).$$

If the variable name x is not included in the current environment $\mathbf{get}(\alpha, \mathit{env})$, the type derivation fails. Otherwise, the lookup function gets the appropriate type v. Obviously, since Lookup does not mention numerical types explicitly, this rule is not affected by the different typing schemes.

Overloading

Numerical operators can be often applied to *different* sets of types of arguments, e.g., + is used to denote addition of two natural numbers, two integers, etc. This form of polymorphism is known as *overloading*. Overloading can be seen either as different ways to invoke the same operator, or as invocations of different operators with the same name. In the latter case, the choice is made based on the data types of the parameters passed. The type checker should distinguish between different ways the same operator is invoked or between different operators of the same name, and derive an appropriate type for the result of the operator application.

Considering an overloaded operator as a number of different operators allows to express the typing rules for each one of them separately. For instance, addition of two integers can be expressed as:

$$\frac{e_1 ==> \mathbf{int} \qquad e_2 ==> \mathbf{int}}{e_1 + e_2 ==> \mathbf{int}}$$

This solution requires, however, a separate rule for each numerical type, e.g., 3 rules for the no widening typing scheme of Chi 1.0 and 9 rules for the full widening scheme of "Chi 1.5". In general, this solution defeats the type checker flexibility we aim at: addition of a new numerical type requires adding $2n$ new transition rules, where n is the current number of numerical types.

To address this problem we need a more compact way of expressing the relation between different types. Specifically, for addition we need predicates "numerical", restricting the application of + only to numerical types, and "lub" allowing us to choose the resulting type based on the types of the arguments:

$$\frac{\begin{array}{c} e_1 ==> t_1 \qquad e_2 ==> t_2 \\ \mathit{numerical}(t_1) \qquad \mathit{numerical}(t_2) \qquad \mathit{lub}(t_1, t_2, t_3) \end{array}}{e_1 + e_2 ==> t_3} \;(\textsc{Add}).$$

Type widening is explicitly present in the rule for addition in the form of the "lub" predicate, least upper bound with respect to the $\preceq$ relation on numerical types. In the case of no widening (Chi 1.0), the relation $\prec$ is empty and $\mathit{lub}(t_1, t_2, t_3)$ is *true* if and only if t_1, t_2 and t_3 coincide. For full widening ("Chi 1.5") $\preceq$ is a linear ordering and t_3 is the larger one of t_1 and t_2. Finally, Chi 2.0 implies $\mathbf{cnat} \prec \mathbf{cint}$,$\mathbf{cnat} \prec \mathbf{nat}$, $\mathbf{cint} \prec \mathbf{int}$, $\mathbf{cnat} \prec \mathbf{int}$, $\mathbf{cnat} \prec \mathbf{real}$ and $\mathbf{cint} \prec \mathbf{real}$. The relation $\preceq$ is a partial ordering and t_3 should be the least upper bound of t_1 and t_2: $\mathit{lub}(\mathbf{nat}, \mathbf{cint}, \mathbf{int})$ should evaluate to *true*.

Type checking user-defined functions

In addition to operators similar to those discussed in Section 3.5 Chi offers a number of functions, e.g., HD(), TL() and LEN() from Example 3.1. Moreover, unlike operators, new function definitions can be introduced by the user. Hence, it is impossible to have a separate rule for each function, like we have for each operator, and a more generic solution is sought. This generic solution should support (1) functions with an arbitrary number of parameters, (2) any combination of parameter types, as well as (3) explicit and (4) implicit templates as mentioned in Section 3.2. We will first discuss the functions without templates, postponing the discussion of explicit and implicit templates until Sections 3.5 and 3.5, respectively.

In addition to function invocation present in traditional programming languages, Chi supports variables of *function type*. Expressions can be assigned to these variables as long as the type of the expression is a function type. Moreover, these expressions, known as function expressions, can be used similarly to traditional function invocations.

Example 5. Consider the following example (adapted from [57]):

```
1    func f(val x : int) → int=⟦ret x ∗ x⟧
2    model M() =
3    ⟦       var     p : (int) → int
4            ,       fv, pv : int
5    ::
6                    p := f
7            ;       fv := f(1)
8            ;       pv := p(−1)
9    ⟧
```

Line 7 shows the traditional function invocation: f is applied to 1 to compute the value to be assigned to fv. In Line 6 f is assigned to p allowing the developer to use p similarly to f (Line 8). □

Intuitively, to type check $p(-1)$ correctly in Example 5 we proceed in four steps. First we need to provide an appropriate typing for p, which in this case would be $\mathbf{int} \to \mathbf{int}$. Then to derive the type for -1, i.e., **int** (**cint**). If the full widening ("Chi 1.5") or constant widening (Chi 2.0) typing system are used we might need to widen the argument types before applying the function: types of the arguments should *match* the function signature but not necessarily be identical to it. Finally, we report **int** as the type of $p(-1)$.

The MSOS rule Function invocation given below formalizes this intuition. Note that * denotes a sequence of objects of the same kind: if t is a type value, then t^* is a sequence of type values; if e is an expression value, then e^* is a sequence of expression values, etc.

$$\frac{\begin{array}{c} e_1 ==> (\mathit{func}({t_1}^* \to t_2)) \\ {e_p}^* ==> {t_p}^* \qquad \mathit{widen}({t_p}^*, {t_1}^*) \end{array}}{e_1({e_p}^*) ==> t_2} \text{ (Function invocation)}.$$

The first precondition of Function invocation corresponds to the first step: e_1 has to evaluate to some function type in order for invocation to be possible. Next, the types of the parameters are determined and widened to types appearing in the function type of e_1. If these preconditions hold, we can report that the type derived for the function invocation is the one prescribed by the function type of e_1.

Formally, *widen* applied to two sequences of equal length ${t_p}^*$ and ${t_1}^*$ is evaluated to *true* if each element in ${t_p}^*$ is not larger with respect to $\prec$ than the corresponding element in ${t_1}^*$.

The type widening is explicitly present in the form of the "widen" predicate. In the case of no widening (Chi 1.0), this predicate should describe the identity relation between types, extended point-wise to sequences. In the full widening and constant widening case, the "widen" predicate has to use the $\prec$ relation to decide if a certain combination is a correct match. In effect, "widen" serves as a generalization of $\prec$, that accepts sequences as arguments instead of single elements.

Explicit templates

Recall that in the Function invocation in Section 3.5, the first step consisted in evaluating the corresponding function expression. In the majority of the cases, a function expression is simply a reference to a declared function or a function variable, but it can also be the instantiation of an explicit template. An explicit function template has a number of template parameters. During template instantiation, the values provided for the parameters are applied to the template, resulting in a new function instance.

$$\frac{\begin{array}{c} e_1 ==> \mathit{func}[{t_{te}}^*, \mathit{id}^*]({t_p}^*) \rightarrow t_r \\ {e_v}^* ==> {t_{tv}}^* \qquad \mathit{widen}({t_{tv}}^*, {t_{te}}^*) \qquad \mathit{eval}({e_t}^*, {t_v}^*) \\ \mathit{replace}(\mathit{id}^*, {t_v}^*, {t_p}^*, {t_{p2}}^*) \qquad \mathit{replace}(\mathit{id}^*, {t_v}^*, t_r, t_{r2}) \end{array}}{e_1\langle {e_v}^*, {e_t}^* \rangle ==> \mathit{func}({t_{p2}}^*) \rightarrow t_{r2}} \text{ (Explicit template).}$$

Type checking an explicit template instantiation is quite similar to function invocation. The topmost step of the rule consists in deriving the type of e_1 to check that it indeed represents a function template. The next step is to derive and check the types of the template parameters. In this version of the rule, we assume that the type parameters are listed last among the template parameters, and that we can distinguish between the type parameters and non-type parameters. Non-type parameters have no further influence on the type derivation, so no further actions are required.

In contrast, the values of the type parameters influence the type derivation. All expressions used for explicit template parameters are required by the language definition to be compile-time constants, so they can be calculated here. The predicate *eval* maps strings representing types ("nat") to actual types (**nat**). The predicate *replace* is applied to get new parameter (${t_{p2}}^*$) and return (t_{r2}) types, where all occurrences of id^* have been replaced by the corresponding value in t_v. If that can be done successfully, we can report that the type derived for the function expression is a function with parameter types ${t_{p2}}^*$ and return type t_{r2}.

Example 6. Recall the example explicit template function type in 3.2:

func $f\langle T$: **type**$\rangle$(**val** a : T) → T. In this case, there are no non-type parameters in the template, so those preconditions are trivially true. In the example, the actual instantiation is $f\langle$nat$\rangle$. The expression nat will evaluate to **nat**, which can replace T in (**val** a : T) → T with no problem. The resulting function type is (**val** a : **nat**) → **nat**. The next step is function invocation, described in Section 3.5.

The type widening present here is similar to the function invocation case. Here, only a subset of the parameters has to be widened, because the type parameters are not numerical and cannot be widened. Apart from that, the widening works exactly the same in each case. Observe that **cnat** and **cint** are not explicitly part of the Chi 2.0 language and do not have a type literal, and, hence, cannot be provided as a type parameter in Explicit template.

Implicit templates

In Example 3.1 we have seen several functions applicable to lists of any type of elements, e.g., HD(), TL() and LEN(). Chi distinguishes between the lists of naturals, lists of booleans, lists of strings, etc. and neither can be considered as a generalization of the other. Hence, if we want the same function to be applicable to all kinds of lists, we would need a separate rule for each one of them. Moreover, providing an element type explicitly would incur an unnecessary overhead on the specification engineer. As this is obviously undesirable, implicit templates were added to the language allowing functions to be defined for variable parameter types without the specification engineer having to name the types explicitly.

The addition of implicit templates makes the rule for function invocation, presented below, significantly more complicated. The function type that results from the function expression can no longer directly be invoked. First we need to determine which function, represented by metavariable e_1, is called: $e_1 ==> \mathit{func}[\mathit{id}^*]({t_f}^*) \to t_{f2}$. Next we need to compute the types of the actual parameters ${e_p}^* ==> {t_p}^*$, so we can find the values of the type parameters with $\mathit{unify}(\mathit{id}^*, {t_f}^*, {t_p}^*, {t_t}^*)$. Finally, we need to replace the type parameters by their values in the return type of the function to which e_1 corresponds.

$$\frac{\begin{array}{cc} e_1 ==> \mathit{func}[\mathit{id}^*]({t_f}^*) \to t_{f2} & {e_p}^* ==> {t_p}^* \\ \mathit{unify}(\mathit{id}^*, {t_f}^*, {t_p}^*, {t_t}^*) & \mathit{replace}(\mathit{id}^*, {t_t}^*, t_{f2}, t_r) \end{array}}{e_1({e_p}^*) ==> t_r} \text{ (IMPLICIT TEMPLATE).}$$

Predicate $\mathit{unify}(\mathit{id}^*, {t_f}^*, {t_{p2}}^*, {t_t}^*)$ is *true* if there exists a widening of ${t_f}^*$ unifiable with ${t_{p2}}^*$ and ${t_t}^*$ corresponds to id^* after the unification. Predicate $\mathit{replace}(\mathit{id}^*, {t_t}^*, t_{f2}, t_r)$ is *true* if t_r can be obtained from t_{f2} by replacing type variables from id^* by the respective types from ${t_t}^*$.

This rule is the most affected one by type widening. The majority of the widening takes place in the "unify" predicate. If there is no widening, "unify" can match types and set values for type variables directly. If there is widening, care has to be taken to ensure that the value for the type variables is not chosen too soon, which could incorrectly fail to derive a type for a correct expression. In addition, if the restricted widening is in effect, a step has to be added to prevent the possibility of a function returning a result of type **cnat** or **cint**, even if all parameters are of those types. A possible solution is the introduction of a "widenConst" predicate:

$$\frac{\begin{array}{c} e_1 ==> \mathit{func}[\mathit{id}^*]({t_f}^*) \to t_{f2} \\ {e_p}^* ==> {t_p}^* \qquad \mathit{unify}(\mathit{id}^*, {t_f}^*, {t_p}^*, {t_t}^*) \\ \mathit{replace}(\mathit{id}^*, {t_t}^*, t_{f2}, t_{r2}) \qquad \mathit{widenConst}(t_{r2}, t_{r3}) \end{array}}{e_1({e_p}^*) ==> t_{r3}} \text{ (IMPLICIT TEMPLATE)}$$

The "widenConst" predicate holds if t_{r2} is equal to t_{r3}, with all instances of **cnat** replaced by **nat** and all instances of **cint** replaced by **int**. This guarantees that a function always returns the proper type. For example, if we apply the function HD() to the list [1], we want the type of the result to be **nat** and not **cnat**.

Folds

Folds in Chi exist to allow simple, often occurring loops to be expressed in one straightforward expression or statement. In this section we discuss addition fold expressions, i.e., folds having + as the aggregation function.

Type checking a fold expression roughly follows the steps of the fold evaluation discussed in Section 3.2. A fold expression has three subexpressions, referred to as e_1, e_2 and e_3. First, the type t_{id} of the generator expression e_1 is determined. The result has to be some kind of container, e.g. a list, and the predicate *container* expresses the fact that t_{id} is the type of the elements in the container type t_1. The identifier *id* is then temporarily added to the environment as a variable with type t_{id} (cf. Example 1). The types of guard e_2 and transformer e_3 are derived in this new environment. Finally, the results of applying e_3 to each value will have to be added together, so they have to meet the criteria for addition. In this case, we know the types of all values will be the same, so the "maximum" predicate is not needed, only "numerical". Type widening is not relevant here.

$$\frac{\begin{array}{c} e_1 ==> \alpha t_1 \\ \mathit{container}(t_1, t_{id}) \qquad \alpha' = \mathbf{set}(\alpha, \mathit{env}, \mathbf{get}(\alpha, \mathit{env})[\mathit{id} \to t_{id}]) \\ e_2 ==> \alpha' \mathbf{bool} \qquad e_3 ==> \alpha' t_4 \qquad \mathit{numerical}(t_4) \end{array}}{\langle +, \mathit{id} \leftarrow e_1, e_2, e_3 \rangle ==> \alpha t_4}\ (\textsc{Addition fold}).$$

Example 7. In Example 3 we have introduced the following addition fold: $\langle +, i \leftarrow [1,2,3,4], i > 2, i * i \rangle$. Here, the *id* is i, and the collection e_1 is $[1,2,3,4]$, e_2 is $i > 2$ and e_3 is $i * i$. First, we determine the type t_1 of e_1 to be a list of **nat**. This means that t_{id} is determined to be **nat**, and the environment is (temporarily) updated to register that i is a natural number. Though the specific rules are not detailed in this thesis, it should be obvious that $i > 2$ and $i * i$ evaluate in the enhanced environment to **bool** and **nat**, respectively. This means the fold is valid, and the result type t_4 can be concluded to be **nat**. □

To adapt ADDITION FOLD to the Chi 2.0 typing scheme we need to use the same idea as in IMPLICIT TEMPLATE: *widenConst* should be applied to the resulting type.

Evolution: Summary

As mentioned in the introduction Chi is an evolving language and the typing system of Chi evolves together with the language. In this section we have reviewed three different typing systems considered by the language designers: not using type widening at all (Chi 1.0), using full widening ("Chi 1.5") and using a restricted from of widening, pertaining solely to explicitly mentioned constants (Chi 2.0). We have seen that the only parts of the MSOS transition rules affected by change of the typing system are related to auxiliary predicates implementing widening or related notions such as replacement.

3.6 MMT Prototype Implementation

To assess the feasibility of the MSOS-based approach as well as to provide the language developers with insights in different alternatives considered, we have developed a prototype implementation. We base our prototype implementation of the type checker on the MMT [17], an interpreter for MSOS specifications in Maude [29].

In the notation used by MMT transition $e == > \alpha t$ is written as $\texttt{E=}\{\alpha\}\texttt{=>T}$. Specifically, if α is $\imath$ we write $\texttt{E=}\{\ldots\}\texttt{=>T}$. In theory, MMT supports predicates, but in practice, we often found it easier to rework predicates to transitions. For example, the rule for function invocation as discussed in Section 3.5 becomes:

```
(Exp1 ={...}=> (Exp3 , func(Type1*,Type2))) ,
((Exp*) match (Type1*) as (Type2)) ={...}=> Type3
-- -----------------------------------------------------------
(Exp1 calls (Exp*)) : Exp ={...}=>
            (((Exp3, func(Type1*,Type2)) calls (Exp*)), Type3) .
```

In this implementation, the first line corresponds to $e_1 == > (\ \mathit{func}(t_1{}^*) \to t_2)$ in the original rule Function invocation. A notable difference is that in the MMT implementation we choose to keep the expression together with the types, in order to get a clearer view of exactly which steps the system made to reach its conclusion. The second line corresponds to $e_p{}^* == > t_p{}^*$ and $\mathit{widen}(t_p{}^*, t_1{}^*)$. We combined the type derivation step with the predicate *widen* into one expression, that rewrites to the return type of the function if the invocation is valid.

Using the machinery provided by MMT and Maude the type checker for a representative subset of Chi 2.0 was implemented by the author, with no previous Maude experience, in less than one month. A *representative subset* of Chi 2.0 was chosen jointly with the language designers. It omits some similar operators, e.g., disjunction and $\leq$ while conjunction and $\geq$ have been included.

Developing the prototype type checker implementation allowed us to provide the language developers with a number of suggestions on how the language can be further improved. These improvement suggestions have been accepted by the language developers and will be considered during the design of future iterations of the language. Some of our improvement suggestions pertained to:

- return types for built-in functions, e.g., MAX has been redefined to $(\mathbf{nat}, \mathbf{int}) \to \mathbf{nat}$ instead of $(\mathbf{nat}, \mathbf{int}) \to \mathbf{int}$, as stated in [58];

- syntax readability, e.g., consistency of naming conventions for built-in operators;

- type literals as the first-class citizens, i.e., one can also write **var** t : **type** = **nat**;
- anonymous functions, i.e., λ, currently being “hidden” inside the fold construct;
- generalization of the fold constructs including user-defined (anonymous) aggregation functions.

A disadvantage of the Maude implementation is that it is linked closely to the structure of the input and output trees. In particular, the input tree must be a Maude term of the correct structure, so the rules can be applied to it correctly. In the same way, the output is always another Maude term. This means the other tools in the language tool set must be capable of producing these terms or reading these terms, which could limit our potential choices. The terms also contain little structure information, which can make them hard to read.

3.7 ASF+SDF Prototype Implementation

Once we had completed the first prototype, we decided to create a more extensive implementation as a second prototype, to see if the shortcomings of the first version could be overcome. In the design of this new component, we considered three general ways: the specification can be implemented by hand, executed by an interpreter or translated (compiled) to a different language already supported by an execution mechanism. Note that for the implementation of parsers all three options are used.

In our case study, we specifically want to experiment with type checker generation. We acknowledge that intrinsically, there is little reason to prefer an interpreted type checker over a generated implementation or vice versa. In order to make our transformation, we first needed to identify our source and target language, which should both be textual. As our source language we chose MSDF, as discussed in Section 3.3, because the existing type checker was in that format. This leaves the target language and the transformation technique to be determined.

Target language: Pyke

First, we choose our target language. Since the type checker has to be integrated in the entire suite of the Chi tools, we choose the language that was used for those tools, i.e., Python to ease integration. In addition to a smooth integration with other Chi tools, this allows us to benefit from a direct mapping of the MSOS labels to the concept of Maps native to Python. However, Python does not natively support the backtracking needed for cases when multiple transition rules can be applied. Rather than adding this ourselves, we used Pyke [45].

Pyke is a Prolog-inspired inference engine operating on backward-chaining rules that can interact directly with Python. Backwards-chaining rules are uniquely *identified* by means of labels and consist of the *conclusion*, indicated by `use`, and a (possible empty) set of *premises*, indicated by `when`. Identifiers can be used, e.g., to refer to rules during error reporting. Both the conclusion and each premise is a term.

During the derivation, the Pyke inference engine maintains a set of *goals*. For each goal the engine looks for a rule with the conclusion matching it. If such a rule is found, the goal is replaced by the set of premises corresponding to the conclusion. According to the result of the matching, variables appearing in the goal might be replaced by the actual values.

Transformation

To implement the transformation we opt for ASF+SDF [19]. ASF+SDF is a term-rewriting language successfully applied to implement program transformation and code generation in industrial cases [30,109]. Alternative languages that could have been used are, e.g., Stratego [23] and Tom [8]. A crucial step in the process of generating the type checker is the ASF transformation that constructs the actual type checker code based on the MSDF specification. The result of the transformation is a set of Pyke rules, that, combined with the type-system-independent supporting Python code that provides support functions, like `max`, implement the type system as described in MSDF. The following sections describe how each of the four main constructs occurring in MSDF specifications is translated.

Imports Recall that the main differences between MSDF and MSOS pertain to the introduction of modules and corresponding imports, and declarations. As a preprocessing step preceding the transformation we eliminate the imports by collecting all MSDF modules into a single specification. This step reduces the number of constructs that have to be handled and allows duplicate rules to be removed or combined before the transformation is carried out.

Formulas One of the features that a Pyke knowledge base can have is an initialization section, a section of Python code where additional functions and variables can be declared and initialized. For example, if we wanted to provide a set of initial constants that can be used in the code, we could define this set in the type system by using a formula to initialize a variable, called say "`init-env`", with a map of constant names and their types. The transformation would then generate Python code in the initialization section that sets the variable to the desired value.

Declarations Recall that declarations describe algebraic data types, and the main operation involving them is testing whether a given term matches a certain data type. When transforming MSDF declarations to Pyke we create a separate backwards-chaining rule for each case in a declaration. Hence, for the declaration in line 3–4 in Figure 3.1, we create six backwards-chaining rules.

The rules created are similar and we illustrate only one of them, corresponding to `dot(Id)`:

```
1   labela
2       use Exp((dot,$la))
3       when Id($la)
```

Here, `labela` in Line 1 is the rule identifier. In Line 2 the keyword `use` introduces the conclusion of the rule, while in Line 3 the keyword `when` is followed by the premises of the rule. In this rule there is only one premise. The rule reads "`(dot,$la)` is an expression if `$la` is an identifier". The underlying assumptions are that the AST is represented by means of tuples (in this case the operator `dot` and the variable `$la`) and that `$` prefixes a Pyke variable.

Transition rules Finally, we have to create backward-chaining rules that implement the transition rules of MSDF. Each transition rule is translated to exactly one backward-chaining rule, where the conclusion of the MSDF transition rule corresponds to the conclusion of the backward-chaining rule, and the preconditions of the transition rule correspond to the premises of the backward-chaining rule. Unlike the preconditions of a transition rule, the premises of a backward-chaining rule are processed in a fixed order. Hence, in our transformation we take care that in the backward-chaining rule the input types are checked first, then the preconditions and finally the label is updated and the result is constructed. To illustrate the generation consider the MSDF rule for `dot` (Lines 18–20 of Figure 3.1). The corresponding backward-chaining rule is

```
1   labelb
2       use trans((dot,$ID), $LABEL, 'real')
3       when
4           first
5               Id($ID)
6           $STATENV = $LABEL['statenv']
7           'real' = $STATENV.get($ID)
8           $DYNENV = $LABEL['dynenv']
9           'cont' = $DYNENV.get($ID)
```

Here, `labelb` in Line 1 is a generated rule identifier. In Line 2 the conclusion is introduced, in this case a transition from the parse tree `(dot,$ID)` to the type `real`. Starting from the following line the premises are listed. First, we test the implicit precondition that the argument that is given for the operator must be valid, i.e., that it is an identifier. This check is delegated to declaration rules like the one generated earlier. We use the Pyke keyword `first` here because there can be multiple proofs that `$ID` is indeed an identifier, but we need only one. The keyword `first` is akin to `once/1` as defined in the Prolog standard [32].

The remaining lines consist of two pairs, each implementing one of the explicit preconditions of the rule. In Lines 6 and 8 we select the relevant environment component. In Lines 7 and 9 the actual lookup is done, and the result is compared to the desired value.

Validation

In the introduction, we stated four challenges in tool development for DSLs. In this section we review the challenges and discuss how the challenges were addressed in our two prototypes.

The first challenge required correctness of the tools supporting DSLs to be clear to the domain experts. We addressed this challenge by (1) separating the specification of the type checker (type system) from its implementation and (2) specifying the type checker in a formal language close to the popular SOS, well-known to the domain experts [11]. As SOS is a popular semantics specification approach for domain specific languages, e.g. Erlang [28] and GP [91], we believe that our approach can be applied to other domain specific languages as well.

The second challenge demands the approach to be suited for evolution, and the third one requires support of decision making by language designers. While these two challenges are quite different, our approach allows to address both of them in a uniform way. Recall that the main components of our approach are type system specification in MSOS and automatic generation of the type checker. In Chapter 3 we have shown that MSOS specifications are well-suited both for specification of type checking evolving languages and for decision making support. Automatic type checker generation allows to regenerate a type checker when the type system has evolved. Implementation of the transformation required merely 235 ASF+SDF transformation rules, amounting to 1174 lines of code. The MSDF grammar used has 115 production rules, and the Pyke grammar 280, 234 of which are part of an imported Python grammar. Hence there are only 46 Pyke-specific production rules.

Our final challenge required the generated tool to fit the existing tools available for the DSL. Since the entire suite of the Chi tools is implemented in Python, we have opted for generation of Pyke code.

Lessons learned

We have received a very positive feedback from the designers of Chi. The methodology applied in this chapter has been chosen by them to support type checking of CIF [9], a new generation hybrid system specification language established by the members of the European network of excellence HYCON[4].

In addition to providing a component for the Chi 2.0 tool set, the research effort described above aimed at practical evaluation of generative approach to type checker development.

- Design of an appropriate type system for Chi 2.0 required an iterative process. We expect that this is the case during the design for most DSLs. The choice of MSOS and code generation reduced the time needed to implement each iteration.

[4] The HYCON project used to have a website at `http://www.ist-hycon.org`, but it no longer exists. The successor of HYCON, HYCON2, can be found at `http://www.hycon2.eu/`.

- Because we constructed the transformation that generates the code to implement the type checker from scratch, we were free to select a target language that suited our needs. By selecting Pyke, a knowledge engine based in the same language the other tools are written in, Python, we could benefit from logic programming capabilities without introducing a number of new dependencies on external software.

- Pyke is both well-suited for MSOS-based type checker generation due to the presence of rules, and for integration with other tools, due to closeness to Python. We expect other rule-based extensions of popular implementation languages such as, e.g., Jess [46] for Java and CLIPS for C, to be well-suited for MSOS-based type checker development as well.

However, we also noticed that smooth integration of the generated type checker with the Chi 2.0 tool set required the type checker to be aware of the representation of AST and DAST. Adding this knowledge to the transformation that generated the code was not a difficult problem, but it resulted in a transformation that cannot be easily reused for other projects and other target languages.

A second, and bigger, problem to using MSOS is that while the language can describe a lot of type system concepts in a convenient and clear fashion, it cannot handle ambiguities very well. Because MSOS is primarily intended for dynamic semantics, MSOS rules can be seen as describing potential execution paths. If a MSOS semantics specification is ambiguous, this can be interpreted in a dynamic setting as a choice between two possible executions, and is quite suitable. In static semantics, however, it is usually expected that an expression has only one valid type in a completed type computation. Thus, having multiple possible type derivations for the same expression is undesirable, especially because MSOS provides no easy, sure way to even detect that there are choices to be made. For example, if we add a rule that results in an error if multiple types can be derived for the same expression, the implementation can simply ignore that rule and produce an incorrect result. A solution to this is to require that the MSOS rule for an expression can only ever give that expression one type, thus preventing ambiguities from forming. This also, however, makes it much harder or impossible to express some common type system features, like overloading. The most natural way to implement overloading in a (MSOS) type system is precisely by using an expression with a different type for each option. Other rules can then restrict this type based on the types of other expressions and the semantics of the language.

A possible compromise is to make each MSOS rule link each expression to a set of types instead of a single type. This way, we can have a single type derivation, while still allowing for multiple types for each expression. The main disadvantage of this approach is that a second step is required, after these sets have been computed, to eliminate them again to get the desired single type values. Usually, this will only involve extracting the only element from a set, but when the set is larger, a selection must be made. We feel this cannot be expressed efficiently in MSOS, and the formalism essentially fails the implicit test in RQ 1.

3.8 Related work

Numerous ways of specifying a type system can be found in the literature. For example, type systems can be described using *denotational semantics* [24, 96]. Denotational semantics is based on functions that give the meaning or denotation of a program by linking inputs to the appropriate outputs. In this case, each node of an abstract syntax tree is linked to the appropriate type. Alternatively, [88] uses Structural Operational Semantics (SOS) to represent type systems, which was also suggested in earlier [90, 105] and more recently in [43]. Unlike denotational semantics, *operational semantics* gives a meaning to terms by defining an abstract machine consisting of states connected by a transition relation. In SOS, the transition relation is defined by sets of rules allowing a step from one state to another if the rule's preconditions are met. When used in type systems, the initial state is the AST, and the resulting state should be the desired type, or a DAST with type information in some or all nodes. Another form of operational semantics is evolving algebras. An *evolving algebra* consists of two parts, a partial, many-sorted algebra that describes the given program as the initial state, and a set of transition rules that describe transitions that allow state changes. Fundamentally similar in approach, evolving algebras mainly differ from SOS-based formalisms in that they have an imperative-programming style rather then the more set-theoretical style used by SOS. Industrial implementations of type systems have usually been based on *attribute grammars* [1,36,107]. Attribute grammars consist of grammars where attributes have been defined for some or all nodes, usually in terms of other attributes. During type checking, attributes are evaluated as needed until the values for the desired attributes, like the type of an expression, is known. Finally, one can develop a special *domain specific language* for type systems [51].

We considered all the specification methods above as a basis for our type checker generator. *Denotational semantics* are clean and independent of implementation, but involve heavy mathematical machinery often surmounted by a highly specialized syntax [87]. Minor changes in the language might require a complete rewriting of the specification [83]. Since we target domain specific languages we need a formalism that would be comprehensible for domain experts without background in formal modeling. As rapid evolution is not uncommon for domain-specific languages, the rigidity of the denotational semantics becomes a major problem. *Attribute grammars* are easier to understand, but focus on how to calculate type values, instead of on what the type should be. Moreover, attribute grammars usually require additional effort from the domain experts: even auxiliary data structures need to be implemented through production rules [40]. *SOS* is closer to natural reasoning, but can be very verbose and specifications tend to have high levels of coupling between rules. Also, earlier work using SOS assumes the programming language being analyzed to be fixed and do not address the co-evolution issues. Finally, while developing a separate *domain specific language* is a valid option for type checker specification, the language developed likely cannot be easily reused for other forms of static semantic analysis, because its constructs are focused on type derivation. Moreover, application of such language requires a special learning effort from the domain experts, similarly to the evolving algebras mentioned earlier.

The idea of using a generative approach to construct type checkers can be found, e.g., in [7,76]. Application of this idea to domain-specific languages, however, requires addressing the challenges stated in Section 3.1. These challenges render such approaches as [7, 76] not suited for domain-specific languages as these approaches were not conceived with the issues of language evolution and tool integration in mind. In particular, in [76], languages are divided into features. If two features are not related in a straightforward manner, extra effort is required to combine them. If a new feature is to be added to a languages that already has many features, this is obviously not desirable. TYPOL programs as described in [7] appear to suffer from the same problem, though the original paper does not state this explicitly.

3.9 Conclusion

In this chapter we have presented our experiences with a novel approach to development of formally specified type checkers. We specified the type system for the specification language Chi 2.0 using MSOS and MSDF and generated a type checker in Pyke, first using Maude and then using ASF+SDF. MSOS is formal, well-suited for evolution and close enough to the popular SOS to be understood by the domain experts with limited programming experience. ASF+SDF allows us to generate a type checker and to reuse the generation process when the DSL evolves.

Developing static semantic analyzers for evolving languages requires the analysis techniques to co-evolve together with the syntax and the semantics of the language being analyzed. In particular, this means that the analyses should be flexible enough to accommodate different syntactical and semantical options contemplated by the language designers. In this chapter we have shown that *modular structural operational semantics provides a possible framework for developing flexible semantic analyses.* This shows MSOS is a possible answer to RQ 3. To illustrate our approach we have chosen to formalize typing rules for expressions in Chi, a hybrid system specification language. We have seen that the analysis developed is indeed:

- flexible enough to co-evolve with the changing language (goal **(i)** from Section 3.1 addressed in Section 3.5, as related to RQ 4),
- and that the MSOS transition rules can be implemented with a limited development effort (goal **(ii)** from Section 3.1 addressed in Section 3.6).

The MSOS-based approach advocated in this chapter can be extended in a number of ways. First, the prototype implementation discussed in Section 3.6

can be further elaborated to provide for error reporting. In case MMT cannot rewrite an expression according to the MSOS transition rules, no specific action is taken. Moreover, if desired, the Maude implementation can be used to generate the type checker code in a programming language chosen by the Chi 2.0 developers. To continue benefitting from the flexibility of the approach, automatic transformation techniques, e.g. [74], can be used.

Next, additional semantics-based analyses can be developed for Chi 2.0. One can, for instance, aim at detecting semantical errors in Chi 2.0 programs. Semantics of Chi 2.0 have been previously formalized using SOS [10], but the formalization is not well-suited for evolution. For example, it was found that adding concurrency or references to the functional language ML required a complete reformulation of the language specification [85]. Therefore, we will consider reformulation of the Chi 2.0 semantics using MSOS in Chapter 3.9. Expressing both the language semantics and the semantic analysis rules in the same semantic framework should lead to seamless integration of the two. Moreover, these translated rules can then be used as starting point to define the dynamic semantics in Action Semantics [37]. Such an Action Semantics definition of Chi2.0 allows another way of executing (and checking) the formal semantics of Chi2.0 via the action environment described in [20].

Unfortunately, we also found significant disadvantages to using MSOS as a type system specification language. The first one is actually related to evolution, not of the language semantics but of its toolset. In particular, the strong connection between the type system and AST structure observed in Section 3.7 limits the reusability of type system rules between different implementations of the same language, especially those is using separate frameworks. In our case, the successor to Chi 2.0, a new language called CIF [12] is implemented using the EMF Framework [101] instead of Python. We would like to reuse Chi 2.0 type system rules for CIF, but the different AST structures makes this almost impossible. This means that MSOS is not as useful in answering RQ 4 as we had initially hoped. It also means that MSOS is not the answer to RQ 3 we are looking for. To resolve this issue, we designed our own type system specification formalism, EMF-TL, which is strongly inspired by MSOS. By integrating the language into an existing DSL framework from the start, we were able connect the type system to existing language infrastructure in a much more straightforward way. We will discus EMF-TL in more detail in Chapter 4.

On the positive side, we found the rule-based definition style of MSOS very useful in the context of RQ 5. By combining several rules that each cover a particular case, we can define types for a variety of elements in a modular way. For a purely theoretical description of a type system, MSOS is a very suitable choice. However, we would like to use our type system specification in a more practical way, and MSOS is less suited to that use case. In particular, we would like to combine the modularity in rule definition with greater modularity in terms of in- and output format. As mentioned before, we will discuss the formalism we designed to meet these requirements in Chapter 4.

Chapter 4

EMF/EMF-TL Introduction

In response to the issues we identified with MSOS as type system specification formalism, we decided to develop an alternative formalism, EMF-TL. EMF-TL is a rule-based transformation language linked to the EMF framework. In contrast with MSOS, EMF-TL features the use of explicit constraints to guide the application of rules. In this chapter, we first introduce EMF-TL, using the modeling language SLCO as an example. We then define the formal semantics of EMF-TL through a three-step process. This process is mirrored in the implementation of EMF-TL, which is discussed next. We show that, given certain assumptions, our implementation matches the previously defined semantics.

4.1 Introduction

One of the issues we discovered during our experiments with MSOS was that the type system definition ended up being too closely linked to the structure of the provided input and expected output. This means, for example, that if the language developers decide to change to a different parser, the type system definition has to be updated to remain consistent with it. Similarly, if the developers introduce a new execution option for the language, new type information may be needed. A possible solution would be to introduce a translation that relates the concepts used in the type checker to those created by the parser or used during the execution. This translation becomes quite cumbersome, however, when the language gets more complicated, and makes the type system specification harder to understand. From what we have seen in relation to RQ 2 in Section 2.3, this is undesirable. Also, the close connection to the implementation actually inhibits language evolution as intended in RQ 4, while the potential support for evolution was one of the reasons for selecting MSOS in the first place.

4.2 EMF

Therefore, we choose a different solution, where we try to find a data format that is flexible enough to accommodate both a variety of parsers and a variety of execution methods. The data format we choose is part of the *Eclipse Modeling Framework*, otherwise known as *EMF* [101]. EMF is based on the concepts of *models* and *metamodels*. The basic unit of EMF is the model. A model consists of a number of *model elements* that can have *structural features*. Note that this means that all abstract syntax trees can be interpreted as models. We cannot, however, represent information from any domain in any model by adding arbitrary features to elements. Every element is based on a *class* that describes what features it has and what values they can contain. All elements that are based on a given class are referred to as its *instances*. Classes are contained in metamodels, and each model is associated with exactly one metamodel. If a given model is actually a abstract syntax tree, the corresponding metamodel directly reflects the abstract syntax used to construct it.

When a model element is created, we have to select a class from the metamodel, which determines what features the element has and how they are initialized[1]. Each feature is defined with a name, a class or type of elements it can contain and a multiplicity, that indicates how many elements it can hold. A feature can be either a *property* or *reference*. A property contains a basic value, that has no further representation in the model. In contrast, a reference contains a pointer from the *source* to a specific model element, which we call the *target*. Usually, this will be another element in the same model, but it can also be the same element, or an element in a different model. Some references are of a special kind, known as *containment reference*. A containment reference represents a relation where the target is considered part of the source. One consequence of this notion is that a model element can be the target of only one containment reference at a time. Another that if a source element is deleted from a model, all target elements of containment references in that element are deleted as well.

SLCO

An example of metamodel is shown in Figure 4.1. The figure was created by taking a fragment of the metamodel of the Simple Language for Communicating Objects or SLCO [4]. SLCO was designed as a DSL aimed at systems consisting of several components cooperating to complete a task. An SLCO system consists of objects connected by channels. Each object is associated with a class that defines its properties. In particular, a class defines the ports that allow channels to connect to objects, and state machines that define the actual behavior of the object. SLCO state machines primarily consist of states and transitions, but can also feature guards, triggers and effects. Guards prevent transitions from activating if given conditions are not met. Triggers, in contrast,

[1] There exists a special value in EMF, called *OCLUndefined*, for features that have no other defined value.

force transitions to be taken if a certain event occurs, like the reception of a message. Effects represent the side effects of transitions, like updating variables or sending messages. In all these constructs, expressions play an important role in defining when a guard holds, what message can be received or what message is sent. Because the expressions elements of SLCO also form the foundation of the largest part of the type system, we choose that fragment of the metamodel that represents expressions to be discussed in detail here.

The main class of the fragment is the `Expression` class, shown in the middle of Figure 4.1. This class has one feature, namely a reference named `type` to another object, that has to be of the class `Type`. This is indicated by a line with an arrowhead pointing at the `Type` class. This reference is a containment reference, and this is indicated by a black diamond at the source of the reference. This is because a `Type` element in isolation is not useful, so once the `Expression` element is deleted, the corresponding `Type` element should be removed as well. The `Expression` class is abstract, as is indicated by the italicization of the class name. This means that no actual instances of the `Expression` class can be created. Instead, the `Expression` class has three subclasses, `ConstantExpression`, `BinaryExpression` and `VariableExpression` as indicated by lines with open arrowheads. Any instance of those classes is also considered an instance of the `Expression` class, and has all the features defined in it. However, the `ConstantExpression` class is itself abstract, and has three subclasses, `BooleanConstantExpression`, `StringConstantExpression` and `IntegerConstantExpression`. These represent the simplest forms of expressions in SLCO, namely direct boolean, string and integer values respectively. In all cases, the actual value is stored as a property called `value`.

A more complex expression is the only other non-compound expression, `VariableExpression`. This class represents references to variables elsewhere in the model, that are represented by the `Variable` class. The connection between these model elements is represented by a reference called `variable`. Note that this is not a containment reference, because a variable can be referenced multiple times, and can exist independently of any references.

The final subclass, `BinaryExpression` represents the only kind of compound expression in SLCO. It represents one of several binary operators, that is applied to the results of two subexpressions. The operator is stored in a property that is based on the `OperatorEnum` enumeration. This is an EMF method to describe properties that should only have one of a small set of values, like the limited set of SLCO operators. The two subexpressions are represented by references to Expression objects. This means a subexpression can be a constant, a variable reference or again a binary expression. Both subexpressions are containment expressions, because they represent parts of the binary expression.

As mentioned before, the `Type` class is instantiated to represent type information. In EMF, type information of the kind computed here is stored in the model like any other information, as elements with properties. In the case of SLCO, each element with a type will contain a `Type` element with a property `primitiveTypeEnum` that contains the specific type. The possible types are defined by the `PrimitiveTypeEnum` enumeration, in the same way as the possible

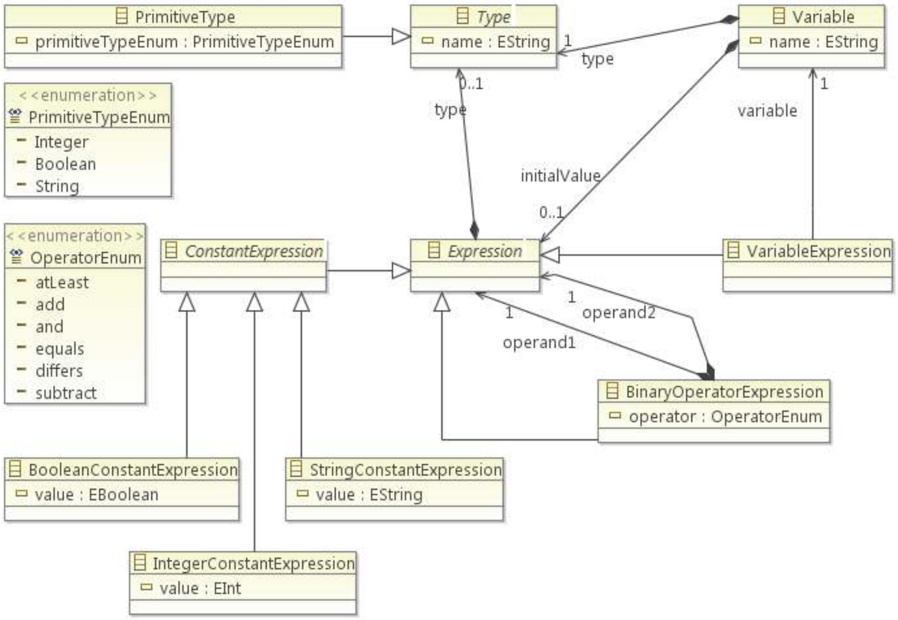


Figure 4.1: Example Metamodel Fragment

operations are defined for binary expressions.

The final class in the metamodel fragment is the `Variable`, that is used to represent the actual variables in the model. Noteworthy here is that because variables are always accessed via references, they do not have a property that represents an identifier. Instead, the only features of a variable are a containment reference to a type, that defines the values the variable can store, and a containment reference to an expression, that determines the initial value of the variable.

Even though it is in principle independent of programming languages, EMF is most closely associated with Java. Based on a metamodel, it is possible to create a set of Java classes that can be used to create and represent models based on that metamodel. Additionally, we can write models to disk using a standard component, in either an XML-based or a binary format. More importantly, there are a number of other components that also use these classes to create and manipulate models. This guarantees compatibility between these tools. Most interesting to us, these tools include parsers, which can be used to create input models for the type checkers and model transformation engines, which can be used both to implement the type checker, and to use its output. Additionally, there are visualization tools, which can be used to create graphical representations of models.

4.3 DSL for type system specification

Based on our experiences with MSOS, described in Chapter 3, and our decision to use EMF as our framework, we decided to create a new DSL to specify type systems, called EMF-TL, inspired by MSOS and our previous work with it, our previous experience with ATL [64] and our experience in creating a specification for the type system of the Java language. We have implemented EMF-TL in Java, based on the Eclipse Modeling Framework, and our implementation is available through a GitHub repository [78].

Like MSOS and ATL, EMF-TL uses rules that are applied to language constructs and define under what conditions it is correctly typed. Unlike ATL, but like MSOS, EMF-TL allows multiple rules to apply to the same element simultaneously during the typing process. Only at the very end of the typing process we require that the number of possible solutions is reduced to one. This way, we enable more use of case distinction in defining type systems, making the definition of complex typing behavior easier and cleaner. It also removes the impact of the order of rule application on the computed types, which in turn removes the burden on the type system designer to consider the order of the application during rule design.

In order to further reduce the impact of order, we have chosen not to introduce an explicit notion of environment in EMF-TL. Environments are normally used to allow model elements that are spread through a model, like declarations and references to those declarations, to be linked together. In EMF-TL, we assume instead that a separate scoping component has connected references to potential targets, so the environment is encoded in the input model. By splitting off scoping from typing, we can simplify the type rules because no environment has to be constructed or manipulated. While this decision could potentially limit the kinds of DSL type systems our language can describe, we feel the simplifications it allows are worth it. Overall, this leads to the sequence of steps shown in Figure 4.2, where a text representation of a model is first parsed, then scoped, and then typed, which leads to the final model. In that process, an EMF-TL type system serves as input for a generator that can create a type checker based on a declarative specification.

Type checking rules

As described in Section 4.2, the EMF framework is based on the concept of models. As mentioned before, however, EMF-TL is based on the concept of rules which apply to *model elements* instead. At this level, in its most basic form, type checking a model element means determining whether it is consistent with a type system. A more complex view is that a type system classifies language elements according to the values they can produce [88] or handle, meaning the result not only states if the language element is valid, but also gives an indication about its behavior. The classes assigned to language elements are called *types*.

A common way to specify this classification is in the form of rules, often called *judgements*. A judgement relates a specific form of language element to a

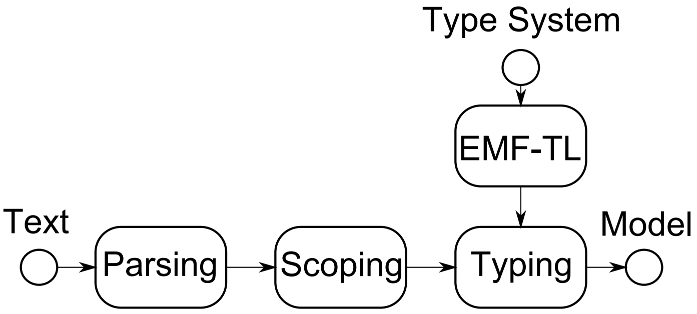


Figure 4.2: Overview of EMF-TL Process

type, possibly with additional conditions on the types and structures of related language elements. This means that types are kept separate from the expressions they describe. In practice, we observe that information computed during type checking, like the types themselves, may be useful not only during type checking itself, but for any tool that uses the model. For example, a code generator might use type information to decide how and where to store intermediate results of computations.

Thus, we believe the result of type checking of a correct model should be a copy of the model extended with information computed during type checking. In this view, a type checker is a model transformation that primarily inserts new attributes into model elements, but preserves the structure of the model. It also means that any type we want to represent in the output model needs a class in the relevant metamodel to represent it, but we feel this is only a minor limitation.

We choose to base the execution of our model on constraint solving to increase the flexibility of our specifications. Each rule in an EMF-TL specification defines the transformation of elements that are an instance of one source class to instances of a target class. Each rule can have a number of conditions attached, that govern where it can be applied, and the properties of the target. A noteworthy point is EMF-TL does not require that a rule fully specifies the desired result. A rule can specify, for example, that a value of a certain property in the result must be smaller than a given value, but not what the value actually should be. During transformation, other rule applications can place additional constraints on the value, which may reduce the number of possible results. If the number of possible solutions is reduced to 0, the condition no longer holds. If it is reduced to 1, we have the desired answer. If there are still multiple possibilities after all rule applications have been handled, EMF-TL uses a so-called strategy to determine of one of the possible result is “better” than the rest. What “better” means is defined by the type system designer. If there is one superior answer, that is the final result. Otherwise, we have a true ambiguity, that should be reported as such. For example, if we can derive both a numeric type and a string type for the same expression, in many type systems we would not know which to choose. In that case, it is better to produce an error that present the user with potentially unpredictable results.

Scoping

As mentioned before, a decision we made in the design of EMF-TL is that we chose to limit the support for scoping. In EMF-TL, rules primarily refer to direct properties of the element the rule is applied to. There is no separate notion of environment that is shared between rules. If we want to type a reference expression, this implies that the intended target of the reference has to be linked to it before it is typed. These links can be created by the parser, or by a separate transformation, for example based on the Name Binding Language [70]. If this is the case, then typing a reference becomes easy, because we can easily access the target of the reference to get any information we need.

In cases where we need type information to choose between different elements as potential reference targets, this also becomes much easier if we have direct access to all candidates. Unfortunately, this type of link cannot directly be created during parsing. Instead, we use a separate scoping step that creates reference links between references and potential targets.

Naturally, this suggests that a separate language is needed to describe the scope system of a language, as a counterpart to the type system. We consider this an advantage, because scoping and typing have to some extent contradictory properties, which make it hard to combine them into a single language. In particular, scoping typically involves handling a number of elements that represent potential targets and scopes and making a choice among them, for example selecting the target of a variable reference from among all variable declarations in the model. These elements can come from anywhere in the model, or even from outside the model. In contrast, type inference or checking involves a limited set of elements that are directly linked, and can to some degree be handled separately. For example, the type of a binary operator expression usually depends only on its two subexpressions, which are directly contained in it.

EMF-TL structure

In this section, we will discuss the structure of EMF-TL specifications. Listing 4.1 shows a fragment of such a type system specification. The full specification can be found in Appendix A. For ease of reference, the line numbers used here match those in the appendix. Each line is numbered as it appears in the actual source file.

EMF-TL Header

In the `imports` section the designer of this type system indicates the metamodels that describe the input and output models. At this point, the prototype implementation of the language does not enforce one to be the input metamodel and the other to be the output metamodel. It is up to the type system designer to ensure that all created elements are compatible with the desired output metamodel, otherwise an error will occur. In fact, the order of the import statements is ignored. In principle, the metamodel for both models can be the same, in that case, only one import statement is necessary.

The next section is the `start` section. In this section, the type system designer should indicate which model element serves as the primary container of the other model elements. There should be exactly one such element in each input model.

Listing 4.1: SLCO EMFTL Header

```
imports
    http://www.emftext.org/language/textualSLCO;
    http://mdse.tue.nl/slco;
start textualSLCO::Model

typesystem
slcotype(type) =
  slco::PrimitiveType(primitiveTypeEnum);
widening
```

In the `typesystem` section, the designer can introduce names for the types to be used in the type system, and associate the types with their representations in the models. In Listing 4.1, the type `slcotype` is introduced and linked to the metamodel element `slco::PrimitiveType`. This class can represent a number of more specific types, like integer and boolean. The parameter `type` is used to describe the more specific type, using the `primitiveTypeEnum` enumeration, which describes the possible types. It should be noted that in the model transformation view of typing, types are represented by language elements not intrinsically different from other language elements. By separating types from their representations, we reduce the coupling between the type system specification and the target metamodel, and hence facilitate the co-evolution of metamodels, as identified in [41], and the corresponding type system specification(s). For example, if a type is renamed in the metamodel, we only need to update the association between the type and the element, rather than updating all rules mentioning that type. Additionally, the type serves as a shorthand, because the name of the type will usually be much shorter than the fully qualified name of the corresponding model element. Because types often occur in several rules, this can help making the type system easier to read and write.

Types declared in the `typesystem` section can be used in the `widening` section to define a subtype relation on types. In SLCO, because the number of types is limited, there is also no type widening, so there is no `widening` section in the header. An example of a type system with widening can be found in the next chapter, specifically in Section 5.2. The section header `widening` is required, even though the section itself is empty. If there was type widening, the language designer would describe how the types involved are related in this section.

A consequence of allowing types to be widened is that there can be situations where the type system has multiple correct outcomes for some model elements. This multiplicity can also be caused by multiple rules for the same element with conditions that are not mutually exclusive. For example, when comparing two numbers, we might be able to choose to interpret them as integers, reals or another numeric types, depending on what types exist in the DSL. The type checker, however, needs to select one “best” type from among all correct types in order to be able to generate a single output model. In other type inference approaches [88], the most general or *principal* type is often considered to be the most desirable one: all correct types are instances of the principal type that can be obtained by instantiating *type variables* present in the principal type. In our case, we assume types to be elements of the input metamodel, i.e., we cannot assume that the type system contains type variables and selecting the principal type may not always be possible. Instead, we have chosen to allow the designer to express a global preference of one type over another. This is defined in the `strategy` section, in the same way as the partial order defined in `widening`. When confronted with multiple types for one element, the type checker prefers the “smallest” type according to the strategy. Again, because SLCO does not feature type widening, nor rules with conditions that are not mutually exclusive, there is no need for a defined strategy. More complex type system, like the one presented in Section 5.2, usually do need strategies. An example of a strategy can be found in Section 5.2.

EMF-TL Rule Conditions

In order to apply a strategy, we first need to have some actual types to compare. We get those by applying the rules defined in the `rules` section of the EMF-TL specification. Each rule states for a specific metamodel element what conditions have to hold for it to be correctly typed. Note that due to the many uses of the word “type”, such as in architecture, biology and typesetting, we cannot assume that every property with the name “type” represents a type that needs to be or even can be computed. Thus, for all elements that are not mentioned in any rule, we assume that they are correctly typed by default and do not need to be changed. The basic structure of a rule is shown in Figure 4.3. In the `from` part, the designers can define what class the rule applies to, and what features they are interested in. When the type checker tries to apply the rule, the variables of the `from` part are initialized with the values of the attributes from the input model. Additionally, any variables defined in the `in` section of the rule are initialized to the closest container of the element that is of the corresponding class, based on the tree structure that is embedded in the model. For example, if the element represents a part of a larger structure, like an object, these variables can be used to access that larger structure. The optional `with` part introduces local variables. The `to` part states how the element will be represented in the output model, and what properties must be updated with computed values. Finally, each rule can have a set of preconditions that govern when it can be applied. Note that multiple rules can be applied to the same class. For instance, a class can represent binary expressions, while different preconditions can restrict the application of the rules to different operators. The rules need not be mutually exclusive. If multiple rules can be applied to an element, but with different resulting types, the strategy is used to choose the minimal solution. If that is not possible, the element is treated as an ambiguity and reported as an error.

```
from MetaModel1::Class1 in (var : MetaModel::Class)*
    ((feature = var)*)
with var*
to MetaModel2::Class2
    ((feature = var)*)
when cond*
```

Figure 4.3: Basic rule structure

```
exp1 <= exp2
exp1 = exp2
exp1 # exp2
exp1 => exp2
exp1 in exp2
for (var(= val)?)* : (cond)*
```

Figure 4.4: Conditions

As shown in Figure 4.3, an important part of a rule in EMF-TL are the conditions that determine when it can be applied. Figure 4.4 shows the six forms of conditions supported by EMF-TL. The first type of condition are binary conditions, shown here as the first five rows. A binary condition combines the values represented by two subexpressions, in a way selected by the operator. The first four operators listed are comparison operators, that hold if the left-hand side is smaller than, equal to, not equal to and larger than the right-hand side respectively. The in operator holds if the left-hand side is an element of the right-hand side. This implies that the right-hand side expression must be a list or a set.

The second type of condition is a for-condition, which can be used to state that a condition must hold for all elements of one or more lists separately. All lists must be of equal length, and the conditions are applied one element per list for each iteration. For example, if we want to define a rule for a container literal, we can use this to apply a condition to each element separately. In another example, we may want to require that each element of the collection has the same type. Another possible use of for-conditions is to compare the types of a list of arguments to the list of parameters expected by a function. A more extended example of the latter can be seen in Listing 5.5. In the for-condition, a variable is declared for each list. During the ith iteration of the for-condition, this variable represents the ith element of the corresponding list. We can also declare a loop variable without an associated list. We assume such variables are intended as a local variable, where the designer is not interested in the resulting list after the loop is finished. These local variables can be used to simplify complicated expressions by splitting them into several parts. For example, the value of an attribute can be stored in a local variable, which can than be referenced from another condition.

number
"string"
`OclUndefined`
var
`Set{`***exp****`}`
`List{`***exp****`}`

exp1 $(+|-|*|\&)$ *exp2*
(length | first | last | flatten | pairs) *exp*

type((*attribute*)*)
MetaModel1::ModelElement1((*attribute* = *exp*)*)

exp1.attribute

Figure 4.5: Expressions

EMF-TL Expressions

In conditions, expressions are used to compute the values we need. Figure 4.5 shows the expressions that exist in EMF-TL. The list starts with basic expressions that are used to represent numbers, string, undefined values and references to variables, respectively. The following expressions allow sets and lists of values to be created. This is useful in combination with the `in` condition operator, in cases where several different values of a property still have the same semantics. In the example SLCO type system rules we show in Listing A.1, we use this to combine rules for binary operators that have the same type of behavior.

The next type of expressions are binary expressions. There are four binary expression operators in EMF-TL, addition (+), subtraction ($-$), multiplication ($*$) and concatenation (&). Note that concatenation is intended for lists and sets, and does not work on strings. There are also a number of unary operators, used in the next set of expressions. They are all related to lists, and they can be used to determine the length of the list (`length`), the first (`first`) and last (`last`) element, collect the elements of a list of lists into one list (`flatten`) and create a lists of pairs of consecutive elements of a list (`pairs`). The last operator is intended for situations where elements of a list form a chain, where for example the output of one expression is the input of the next. By splitting the list into pairs, we can set conditions for each connection in the chain separately, using a for-condition. While there are of course many more operations on lists that could be included in EMF-TL, we choose these based on our previous experience with describing type systems.

The next two expressions describe patterns, the first for types, and the second for model elements. Patterns can be used to restrict elements to specific structures, and to extract the values of properties from types and model elements. One limitation is that model element patterns can only use values from the input model. Patterns do, however, support inheritance. If the element we try to match is actually linked to a subclass of the defined class, the match can still succeed. If we want to access values computed by the type checker, we have to use the last expression. Using that, we can both extract property values from arbitrary elements, and access values computed during type checking. A common use for this is to get access to the types of subexpressions in order to compute the type of a compound expression.

EMF-TL Specification Example

We now return to the SLCO type system to show examples of actual EMF-TL rules. The first three rules in the type system specification are shown in Listing 4.2 in lines 12 to 22. These are also the simplest rules one is likely to find in a type system definition. In each case, the input element contains a constant basic value that belongs to one specific type. The target element of the rule is essentially the direct equivalent of the source element in the target model, and the only thing that needs to be done is to assign the appropriate type to the corresponding attribute of the target.

Listing 4.2: SLCO EMF-TL Rules

```
10  rules
11
12  from textualSLCO::StringConstantExpression
13  to slco::StringConstantExpression(type = $t)
14  where $t = slcotype{{type =
      slco::PrimitiveTypeEnum::String}}
15
16  from textualSLCO::BooleanConstantExpression
17  to slco::BooleanConstantExpression(type = $t)
18  where $t = slcotype{{type =
      slco::PrimitiveTypeEnum::Boolean}}
19
20  from textualSLCO::IntegerConstantExpression
21  to slco::IntegerConstantExpression(type = $t)
22  where $t = slcotype{{type =
      slco::PrimitiveTypeEnum::Integer}}
```

In SLCO, binary expressions are all represented by one class of elements, `textualSLCO::BinaryOperatorExpression`, with an attribute, `operator` that indicates which operation is used in this expression. Two additional attributes, operand1 and operand2, represent the two subexpressions. Because the different SLCO operators have different semantics, the binary expression elements are covered by a group of four rules instead of a single rule. In each rule, one of the conditions restricts the value of the operation attribute to just those the rule applies to. As the first two rules demonstrate, this does not mean that the rule only applies to a single operator, but multiple operators can share a rule if they happen to share semantics. In the first rule, shown in line 24 to 28, describing equality operators, the condition on line 27 states that the types of the two subexpressions are equal. In SLCO, if two values are of different types, they must be unequal, so it makes no sense to compare them. If the types are equal, the type of the binary expression is boolean. Note that we use the shorthand defined in the type system section shown in Listing 4.1. In order to indicate the shorthand is used, we use double instead of single curly brackets.

The next rule, shown in line 30 to 36, describes the two arithmetic operations, addition and subtraction. In this case, not only the types of both subexpressions must be equal, both must be of the integer type. If this holds, the type of the expression is again integer. The third rule, shown in line 38 to 43, describes the boolean conjunction operator. It is essentially the same as the previous rule for arithmetic expressions, but for booleans instead of integers. The fourth rule describes a numerical comparison operator. This rule is essentially a combination of the previous two. As with the arithmetic operators, only numbers can be compared this way. As with the boolean conjunction operator, the result is a boolean value.

Listing 4.3: SLCO EMF-TL Rules

```
24  from
      textualSLCO::BinaryOperatorExpression(operator
      = $o, operand1 = $l, operand2 = $r)
25  to slco::BinaryOperatorExpression(type = $t)
26  where $o in set{textualSLCO::Operator::differs,
      textualSLCO::Operator::equals},
27        $r.type = $l.type,
28        $t = slcotype{{type =
            slco::PrimitiveTypeEnum::Boolean}}
29
30  from
      textualSLCO::BinaryOperatorExpression(operator
      = $o, operand1 = $l, operand2 = $r)
31  to slco::BinaryOperatorExpression(type = $t)
32  where $o in set{textualSLCO::Operator::add,
33                  textualSLCO::Operator::subtract},
34        $l.type = $t,
35        $r.type = $t,
36        $t = slcotype{{type =
            slco::PrimitiveTypeEnum::Integer}}
37
```

```
38  from
      textualSLCO::BinaryOperatorExpression(operator
      = $o, operand1 = $l, operand2 = $r)
39  to slco::BinaryOperatorExpression(type = $t)
40  where $o = textualSLCO::Operator::and,
41        $l.type = $t,
42        $r.type = $t,
43        $t = slcotype{{type =
            slco::PrimitiveTypeEnum::Boolean}}
44
45  from
      textualSLCO::BinaryOperatorExpression(operator
      = $o, operand1 = $l, operand2 = $r)
46  to slco::BinaryOperatorExpression(type = $t)
47  where $o = textualSLCO::Operator::atLeast,
48        $l.type = slcotype{{type =
            slco::PrimitiveTypeEnum::Integer}},
49        $r.type = slcotype{{type =
            slco::PrimitiveTypeEnum::Integer}},
50        $t = slcotype{{type =
            slco::PrimitiveTypeEnum::Boolean}}
```

The next rule, shown in line 52 to 55 describes the first reference expression in the type system. As discussed before, a separate scoping step, executed before the type checker, finds and stores all potential reference targets in the `links` property. The rule has only to select the right `Variable` from the candidates. In case of SLCO, as in many languages such as C and Java, this choice is primarily based on the name of the variable, which must match the name given in the reference. This check is done in line 55, where the variable declaration element can only match the given pattern if the value of its `name` property matches the value of the `$varname` variable. From the variable, we can then extract the declared type, which is also the type of the expression.

Listing 4.4: SLCO EMF-TL Rules

```
52  from textualSLCO::VariableExpression(links =
      $scope, name = $varname)
53  to slco::VariableExpression(type = $t, variable =
      $v)
54  where $v in $scope,
55        $v = textualSLCO::Variable(name = $varname,
            type = $t)
```

A similar pattern is used in the other rules in the type system. An example is the next rule, shown in Listing 4.5 in line 57 to 63, which describes how `textualSLCO::Transition` elements should be handled. This element is unusual because it contains not one but two references, namely to both the source and target state. These are both resolved in almost the same way as the variable and class references described earlier. The only difference is that in the case of transitions, to ensure the source and target state are not swapped, we require explicitly that the name of the source state matches the name given in the element, and the same for the target state. For the type system, the only other relevant property is the guard. The guard is an expression, and thus it has a type. Because a guard essentially switches a transition on or off, it has to result in a boolean value. This requirement is represented by the last line of the rule.

Listing 4.5: SLCO EMF-TL Rules

```
57  from textualSLCO::Transition(sourcename =
      $source, targetname = $target, guard = $g,
      links = $links)
58  to slco::Transition(source = $sourcevertex,
      target = $targetvertex)
59  where $sourcevertex in $links,
60        $sourcevertex = textualSLCO::Initial(name =
            $source),
61        $targetvertex in $links,
62        $targetvertex = textualSLCO::Initial(name =
            $target),
63        $g.type = slcotype{{type =
            slco::PrimitiveTypeEnum::Boolean}}
```

Another example of a reference rule is the rule, shown in line 65 to 70, for the assignment statement. When executed, an assignment statement updates a variable to the result of an expression. In order for this to work, the result of the expression has to be of the same type as the variable. This is directly expressed in the rule. First, we resolve the reference to determine which variable is being assigned to. Then, we extract the type of the variable, and compare it to the computed type of the expression.

Listing 4.6: SLCO EMF-TL Rules

```
65  from textualSLCO::AssignmentStatement(expression
      = $e,links = $links)
66  with $t
67  to slco::AssignmentStatement(variable = $variable)
68  where $variable in $links,
69        $variable = textualSLCO::Variable(type =
            $t),
70        $e.type = $t
```

The final set of rules all involve variations on rules introduced earlier. The first four, shown in line 72 to 92, all handle a single reference, to a class, an input port, a variable and an output port respectively. Note that ports, signals and channels are all untyped in SLCO. In other words, any message can be sent or received through any port, and any two ports can be connected via a channel. This means a textualSLCO::SendSignalStatement or a textualSLCO::SignalReception can never be incorrectly typed, as long as they use an existing port in the right class. The last rule, shown in line 94 to 103, handles elements of type textualSLCO::Channel. Each SLCO channel connects two objects via one port each. Thus, a channel expression contains four references, one for each object and one for each port. As one might expect, the conditions for the rule for channels essentially consist of four copies of the conditions for simple references.

Listing 4.7: SLCO EMF-TL Rules

```
72  from textualSLCO::Object(links = $links)
73  to slco::Object(class = $class)
74  where $class in $links,
75        $class = textualSLCO::Class
76
77  from textualSLCO::SignalReception(links = $links)
78  to slco::SignalReception(port = $port)
79  where $port in $links,
80        $port = textualSLCO::Port
81
82
83  from textualSLCO::SignalArgumentVariable(links =
      $scope)
84  to slco::SignalArgumentVariable(variable = $v)
85  where $v in $scope,
86        $v = textualSLCO::Variable
87
88
89  from textualSLCO::SendSignalStatement(links =
      $links)
90  to slco::SendSignalStatement(port = $port)
91  where $port in $links,
92        $port = textualSLCO::Port
93
94  from textualSLCO::Channel(links = $links,
      object1name = $object1name, object2name =
      $object2name, port1name = $port1name, port2name
      = $port2name)
95  to slco::Channel(object1 = $object1, object2 =
      $object2, port1 = $port1, port2 = $port2)
96  where $object1 in $links,
97        $object1 = textualSLCO::Object(name =
            $object1name),
98        $object2 in $links,
        $object2 = textualSLCO::Object(name =
          $object2name),
        $port1 in $links,
        $port1 = textualSLCO::Port(name =
          $port1name),
        $port2 in $links,
        $port2 = textualSLCO::Port(name =
          $port2name)
```

4.4 EMF-TL semantics

EMF-TL is designed to be a flexible language for defining type systems. The goal of every type system is to find types such that all type system rules are satisfied. We would like to prove that if such values exist for a given model and EMF-TL type system, our implementation of EMF-TL will find them and produce a correct output model. Furthermore, we would like to prove that our implementation will never produce an output model based on values that do not fit the constraints that result from the type system. In order to do this, we need a more formal description of the semantics of EMF-TL.

The first step, the *constraint phase*, computes how type system rules can be applied to model elements. As elements are processed, each rule that could be applied leads to a set of constraints. The constraints encode the conditions of the rules. If we arrive in a situation where there are no constraints left and each variable has a value, we have a solution. Once we reach a solution, we know the model can be typed, and the result of this step is for each model element the rule that was used to type it, and the values of the variables involved in that solution. On the other hand, if we arrive at a set of constraints where there are no valid values for at least one variable, we know that no solution can be reached, and at least one rule application must be invalid. We will define the semantics of the constraint phase in Section 4.4.

This leads to the second step, *model generation*. In this step, each input model element is considered again. Based on the results of the constraint phase, a corresponding output model element is created for each input element. If our assumptions on the relevant metamodels hold, this will result in an output model that contains all information of the input model, enhanced with type results. We do not discuss the details of model generation here, but we do discuss the broad details in Section 4.4.

Constraint semantics

The constraints part of the EMF-TL semantics is defined using a *constraint logic program* [62]. A constraint logic program is based on *constraint variables.* Constraint variables act like variables in both logic and imperative programming in that they can represent plain values, but they can also have domains. A domain is fundamentally a set of values. In this case, the values are either

EMF elements, types or atomic values, like integers or strings. If a variable has a domain, it can only represent values from that domain. Domains can be added to variables or modified by using *constraints*. In a constraint logic program, *constraint handling rules* can be used to define how a constraint affects the domain of a variable. We use such constraint handling rules to define the semantics of EMF-TL type rules and conditions. In this section, we first discuss the global structure of the translation from rules to constraints. Following that, we will discuss the details of individual constructs.

Global structure

To describe how we relate EMF-TL rules to constraint handling rules, we first need to describe which constraint variables and constraints we use. We start with constraint variables, which we divide into two kinds, *global* and *local*. During construction of the constraint program, global constraint variables are defined for each model element. These variables are accessible throughout the program, in contrast to local constraint variables, and used to expose computed results to other rules. One of the variables for each element concerns the specific rule that is applied to the element. Recall that only one rule can be applied to each element in a valid solution. In a complete constraint solution, this variable should contain the identifier of exactly one EMF-TL rule, which will be used to create a model element in the output model. As an example, consider an SLCO `StringConstantExpression` element. If the element can be correctly typed, there must be a rule for which all conditions are met, which in this case must be the first rule shown in Listing 4.2. Thus, the identifier of that rule will be recorded in the constraint solution to be used later. The other global variables represent the properties that are computed during constraint solving. Using these properties, rules can access the results of computations done for other elements. During the constraint computation, we also use local variables. Local variables are variables related to specific rules, and are not accessible outside of that rule. Local variables are added to a separate variable store as rules are applied.

Once we have constraint variables, we can define constraints. In its most basic form, a constraint consists of a name and some variables. We write this down as ”name(V)”, where ”name” represents the name and V represents the collection of variables. In our system, a constraint has no innate meaning, and exists purely as an object that can be manipulated by constraint handling rules. Constraint handling rules remove one or more constraints and implement the semantics of those constraints by either changing the domains of variables, introducing different constraints, or both. When the program is started, it is initialized with a set of constraints that represent the elements to be typed. By applying a sequence of constraint handling rules, the goal is to arrive in a state where all constraints have been removed. At that point, all information about the variables from the constraints is transferred to the domains. If no domains are empty, this means we can construct valid solutions by assigning each variable a value from its domain.

Thus, in short, we want to compute the domains of the constraint variables such that all values in the domain represent correct type system results. Obviously, the nature of these domains depends on both the input model and the type system specification. In particular, since we have one rule-recording variable per model element, we also have multiple alternative implementations of the same rule for different model elements. Each implementation encodes the relevant information from both rules and model elements in the constraint handling rules. In the resulting constraint program, we have a constraint handling rule for each combination of model element and type rule that can be applied to it, based on the source part of the rule. In theory, this can result in a number of constraint handling rules equal to the number of model elements multiplied by the number of type system rules. Indeed, if the metamodel of the input model contains only unique class, this is what will happen, because all type rules will apply to all model elements. In more practical scenarios, the number of constraint handling rules is much lower.

Single rule structure

Each of the constraint handling rules in our constraint program has the structure shown in Example 8. In the example, the rule(V) construct represents the constraint this rule handles by translating it into other constraints. Each instance of a rule is parameterized by references to a subset of the global variables, V, related to one element. I are equations initializing local variables with information from the input model, which is globally availably but read-only, and B are the guards of the rule from the specification. The equation $v_1 = \texttt{Rulename}$ records that this rule is applied, using the global constraint variable v_1 taken from the set V and the rule identifier `Rulename`. Finally, the constraints in O relate the results of the conditions to the global constraint variables V representing properties of the output element. In the rules, we will use $x, y, \ldots$ as names of variables, $\phi, \psi, \ldots$ as names of values and attributes, and capitals of these letters, $X, Y, \Phi, \Psi, \ldots$ as names for sets of variables and values or attributes respectively.

Example 8. $\text{rule}(V) \Rightarrow I, B, v_1 = \texttt{Rulename}, O$

Example 9. Consider, for our first example, the rule for strings in Listing 4.8. This is one of the simplest possible type system rules. This rule applies to elements of class `textualSLCO::StringConstant` and has only one condition, which states the type of a string constant is always `textualSLCO::String`. An example of a model element this rule could apply to is the following:

```
"text"
```

Listing 4.8: SLCO EMF-TL Rules

```
from textualSLCO::StringConstantExpression
to slco::StringConstantExpression(type = $t)
where $t = slcotype{{type =
   slco::PrimitiveTypeEnum::String}}
```

In order to set the type of this element to the right value, we need to create a constraint handling rule that implements the type rule for strings mentioned earlier. Even for a simple model element like this, we need a separate constraint handling rule for each element representing a string expression in the model. A typical instance of such a constraint handling rule is shown below. Compared to Definition 8, this rule has no input variables, so initialization I is empty. The expression t=slcotype{{**type**= String}} in the rule represents the body B of the rule. We next note the global variables, v_1 and v_2. Variable v_1 will eventually contain the identifier of the rule used to type the string expression element and has a domain consisting of all possible rule identifiers, and variable v_2 will contain the type of the element the rule has been applied to and has the domain of all possible types. We assume that `type` is the only property computed in this type system, so there are only two global constraint variables per element. The expression $v_1 =$ "StringConstant" corresponds to the $v_1 =$ `Rulename` part. Finally, the expression $v_2 = t$ implements the output part O by copying the value of t to variable v_2. Note that, for the purpose of readability by reducing the sizes of rules, we drop the metamodel prefixes of the class references, like `textualSLCO::` from `textualSLCO::StringConstant`, from now on, because they are not relevant for this example.

$$\begin{aligned} \text{rule}(v_1, v_2) \quad \Rightarrow \quad & t = \text{slcotype}\{\{\textbf{type} = \mathit{String}\}\}, \\ & v_1 = \text{“StringConstant”}, \\ & v_2 = t \end{aligned}$$

In the constraint handling rule, these relations are represented by a number of equalities. The first one essentially sets the value of v_1 to the identifier of the rule used to type this element. Note that, in practice, this identifier is likely to be generated and has no actual connection to the semantics of the rule, but for the purpose of this example, we choose a more meaningful name. The second equality, which corresponds directly to the condition of the type rule, sets the local variable t to the desired type of the element. The final equality relates the value of the global variable v_2 to the local variable t, indirectly also updating it to the desired type. This way, if computations on other elements would require information on the type of this element, the rule being evaluated can acquire the information it needs by accessing this variable.

Example 10. Consider, for our second example, the rule for the equality and difference operator of SLCO, as shown in Listing 4.9. These rules were shown before in Listing 4.2, and reproduced here for readability. In this rule, lines 24 and 25 describe the transformation part of the rule, the source and target element. When the SLCO EMF-TL specification is applied to a model, this rule can potentially be used to calculate the type of any `BinaryOperatorExpression` in that model. Lines 26 to 28 describe the constraint part of the rule, the conditions that need to be satisfied if this rule is indeed the one that will be used.

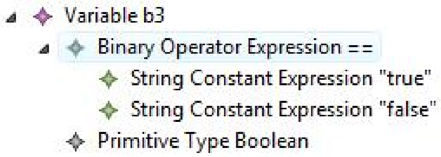
Listing 4.9: EMF-TL semantics example

```
from
  textualSLCO::BinaryOperatorExpression(operator
  = $o, operand1 = $l, operand2 = $r)
to slco::BinaryOperatorExpression(type = $t)
where $o in set{textualSLCO::Operator::differs,
  textualSLCO::Operator::equals},
      $r.type = $l.type,
      $t = slcotype{{type =
        slco::PrimitiveTypeEnum::Boolean}}
```

In an actual SLCO model, the equality operator could be used like this:

```
Boolean b3 = "true" == "false"
```

Here, we use the equality operator to compute an initial value for the boolean variable `b3`[2]. While the example is in textual form, a parser will create an ECORE model that consists of separate model elements, including one of type `BinaryOperatorExpression` that represents the equality. This model is shown in Figure 4.6.



```
Variable b3
  Binary Operator Expression ==
    String Constant Expression "true"
    String Constant Expression "false"
  Primitive Type Boolean
```

Figure 4.6: Example in model form

We know that the rule can be applied to the `BinaryOperatorExpression` element based on the class used to define it, but in order to apply the rule its conditions must also hold. In order to determine if this is the case for the example model element, we need to create a constraint handling rule based on the pattern described in Definition 8. In pseudocode, for the rule in Listing 4.9, this results in the state shown below.

[2] Obviously, this is not a particulary efficient way to initialize this variable, but it will serve as an example.

$$\begin{aligned}
\text{rule}(v_1, v_2) \quad \Rightarrow \quad & o = \text{equals}, \\
& l = \text{StringLiteral}(\text{value} = \text{“True”}), \\
& r = \text{StringLiteral}(\text{value} = \text{“False”}), \\
& o \text{ in } \{\text{equals}, \text{differs}\}, \\
& r.\text{type} = l.\text{type}, \\
& t = \text{slcotype}\{\{\textbf{type} = \text{Boolean}\}\}, \\
& v_1 = \text{“Equality”}, \\
& v_2 = t
\end{aligned}$$

In the example, the first thing to note are the global constraint variables v_1 and v_2, where v_1 will eventually hold the identifier of the rule that is applied to this element, and v_2 will eventually hold the type computed for this element. There are also a number of local variables, o, l, r and t. The first three, o, l and r, are defined in the source element of the rule, so they are initialized in the initialization section of the rule. In this example, the subexpressions are two instances of the `StringLiteral` class. The next constraint sets v_1 to the identifier of the rule, “Equality”, thus preventing the application of any other rule to the same element. The next part contains the three conditions of the rule, in turn. In the final part, the global constraint variable v_2 is set to the value of the variable t.

Constraint handling rule application

Using this kind of rules, we define the operational semantics of the constraint part of an EMF-TL specification as a transition system. This transition system models the constraint solver defined by the constraint handling rules created based on the EMF-TL specification and the input model. A step in the transition system represents a decision made by the type checker. A state of this transition system is a tuple of the following form: $< G_s, D_s, L_s >$. In this tuple, the relation $D_s : V \mapsto DV_s$ represents the *global variable store*, which is used to store pairs of global variables from the set of variables V and their computed domains from the set DV_s of all possible domains, as the computation progresses. The global variable store is initialized with a conceptual domain D_u, representing all possible EMF model elements, like for example `textualSLCO::BinaryOperatorExpression`, all possible types, like for example `slcotype`, and all possible atomic values, like `true` or `false`, for each result variable. By definition, all EMF models, and thus all sets of EMF elements we will encounter, are recursive [97], which means membership of these domains is always decidable in finite time. Because the number of model elements is constant during constraint solving, we know how many global variables are needed at the start, and variables are not added or removed during computation. Using the type system from Example 10, the global variable store would consists of two global variables for each of the three model elements. In contrast, L_s contains the local variables of the rules that have been applied and the corresponding domains. Initially, L_s is empty. As more rules are applied and decomposed into smaller constraints, more variables are added to L_s, renamed as necessary to avoid conflicts. G_s is called the *goal store* and contains a conjunction of all constraints. Note that the goal store is assumed to be never empty. If there are no constraints left, we define its value to be simply `true`. Initially, G_s is filled with a conjunction of *rule* constraints, representing the model elements to be typed.

As each model element is different, there is a separate *rule* element for each typeable model element in the model. This will be discussed in more detail in Example 11 on page 99. As constraint handling rules are applied, the rule elements are replaced by more specific constraints, according to the following rule, where V_{other} represents the variables used in rule_i other than v_1, and V_{local} represents the local variables of rule_i:

$$\frac{\begin{array}{c}(G_s = \mathrm{rule}_i(v_1, V_{other}) \wedge G_s^{'}) \wedge \\ (\mathrm{rule}_i(v_1, V_{other}) \Rightarrow I_i, B_i, v_1 = \texttt{Rulename}, O_i) \wedge \qquad L_s \cap V_{local} = \epsilon\end{array}}{\begin{array}{c}< G_s, D_s, L_s > \qquad ==> \\ < I_i \wedge B_i \wedge v_1 = \texttt{Rulename} \wedge O_i \wedge G_s^{'}, D_s, L_s \cup V_{local} >\end{array}} \; (\textsc{Main}).$$

This rule defines a step in the constraint handling process. It can always be applied, as long as there is a rule element in the goal store left to process, hence it has no premise. When applied, it removes one goal and creates new goals, which are instances of EMF-TL language constructs, and creates local variables where necessary. In this formulation, V_{other} represent the global variables other than v_1 which are associated with this element, as described on page 86. For example, in the rule shown earlier, V_{other} is instantiated as $\{v_2\}$. These variables are used in the output constraints O_i to make computed values available for other rules. The set V_{local} contains the local variables of this rule, which have to be added to the constraint store when it is applied, renamed as necessary. As they are added, their domains are also initialized to the set of all possible model elements, because we have no information to eliminate any candidate at this point. We assume that the constraint engine automatically reduces the conditions to a normal form where no expressions are nested, i.e., if a nested expression occurs in one of the conditions, the inner expression is replaced by a fresh variable and an additional condition is created that links the value of the new variable to the old inner expression. For example, if we consider the expression $r.\mathrm{type} = l.\mathrm{type}$, we can see that two projections are nested in an equality. We can split this into $x = y, x = r.\mathrm{type}, y = l.\mathrm{type}$, with fresh variables x and y. This normalization greatly reduces the number of constraint forms we have to consider. We refer to the final result as the *decomposition* of the original conditions.

Constructs: Equality and Comparison

The most fundamental set of rules deals with equality. Because we consider the constraint process finished when no constraints are left, the equality constraints have to be eliminated at some point. We do this by essentially shifting the values from the constraints to the relevant domain store. Because the rules for local and global variables are so similar, we only show the rules for global variables. The rules for the local variables are the same, with references to the global domain store replaced by changes to the local domain store. Recall that all values involved in type computation are either model elements, types or atomic values. We define a specific rule for each case. The most basic rule deals with the atomic values. If we have derived a constraint that equates a variable to a single value, we can reduce the domain of that variable to only that value, as shown in rule Equality I. If this reduces the domain of the variable to the empty set, we know this configuration of constraints has no solutions. Note that we use the $\implies$ symbol to indicate implication.

$$\frac{\begin{array}{c}\forall y(y = x \implies D_s^{'}(y) = D_s(y) \cap \{\phi\}) \land \\ \forall y(y \neq x \implies D_s^{'}(y) = D_s(y))\end{array}}{< x = \phi \land G_s, D_s, L_s > ==> \; < G_s, D_s^{'}, L_s >} \text{ (Equality I)}$$

The next most basic case deals with storing model elements. This rule, Equality II, is very similar to rule Equality I, but deals with a more complex construct. Note that we use Φ here to indicate the set of attributes a model element can have, and P to indicate the corresponding set of values. As an extension of that, we use $\Phi = P$ to indicate the pairwise relation between a set of attributes and a set of values. The result of the step is that the domain of x is reduced to all model elements of type *ModelElement1*, with attribute and reference values that fit with the other constraints. We use the notation $\{z : z = \mathit{ModelElement1}(\Phi = P)\}$ to indicate the set of zero or more possible elements based on `ModelElement1` that fit the current constraints.

$$\frac{\begin{array}{c}\forall y(y = x \implies \\ (D_s^{'}(y) = D_s(y) \cap \{z : z = \mathit{ModelElement1}(\Phi = P)\})) \\ \land \forall y(y \neq x \implies D_s^{'}(y) = D_s(y))\end{array}}{\begin{array}{c}< x = \mathit{ModelElement1}(\Phi = P) \land G_s, D_s, L_s > \qquad ==> \\ < G_s, D_s^{'}, L_s >\end{array}} \text{ (Equality II)}$$

The next rule, Equality III is very similar, but involves types instead of model elements.

$$\frac{\begin{array}{c}\forall y(y = x \implies D^{'}_s(y) = D_s(y) \cap \{z : z = \mathit{type}(\Phi = P)\}) \wedge \\ \forall y(y \neq x \implies D^{'}_s(y) = D_s(y))\end{array}}{< x = \mathit{type}(\Phi = P) \wedge G_s, D_s, L_s > == > \; < G_s, D^{'}_s, L_s >} \text{ (EQUALITY III)}$$

The final equality rule, EQUALITY IV, deals with equality between model elements. Recall that, as defined in Section 4.2, model elements have are instances of a model class, which defines what attributes they have. In order to streamline the metamodel definition, one class can be defined to be a subclass of another, which means it inherits all attributes from the parent class. In computing possible value, we have to take these inheritance relations between classes into account. Inheritance relations create situations where we consider two model elements as being equal, despite the elements belonging to different classes. In such a case, one class must be a subclass of the other, according to the metamodel. This is indicated by the $\lhd$ condition. The two objects do have to match on the attributes of the superclass, Φ. This is represented in the rule by replacing the initial constraints by constraints on the attribute expressions. Only if the values of the relevant attributes are equal, the original elements can be equal. In addition to expressing the comparison, this also creates new equalities which can be used to narrow domains. Note that we use the instanceOf operator to indicate an element is a member of a given class.

$$\frac{\begin{array}{c}\mathit{ModelClass1} \lhd \mathit{ModelClass2} \wedge \\ \{\forall z : (z = \mathit{ModelElement1}(\Phi = P)) \implies \\ (z \,\text{instanceOf}\, \mathit{ModelClass1})\} \wedge \\ \{\forall z : (z = \mathit{ModelElement2}(\Phi = Q)) \implies \\ (z \,\text{instanceOf}\, \mathit{ModelClass2})\}\end{array}}{\begin{array}{c}< x = y \wedge x = \{z : z = \mathit{ModelElement1}(\Phi = P)\} \\ \wedge\, y = \{z : z = \mathit{ModelElement2}(\Phi = Q)\} \wedge G_s, D_s, L_s > \\ == > \qquad < P = Q \wedge x = \mathit{ModelElement1}(\Phi = P) \\ \wedge\, y = \mathit{ModelElement2}(\Phi = Q) \wedge G_s, D_s, L_s >\end{array}} \text{ (EQUALITY IV)}$$

As a counterpart to the equality operator, we have the inequality operator. This operator is used when we want to block a specific value for a variable. Unlike equalities, a inequality constraint actually provides little information on its own. In fact, because a inequality can never provide values that need to be stored, we need less rules to define it than we needed for equality. If the inequality value ϕ is not in the current domain of the variable, the inequality provides no new information at all, so we can remove it directly. If ϕ is in the domain, we can update the domain by eliminating that value.

$$\frac{\begin{array}{c}\forall y(y = x \implies D_s^{'}(y) = D_s(y) \setminus \{\phi\}) \wedge \\ \forall y(y \neq x \implies D_s^{'}(y) = D_s(y))\end{array}}{< x \neq \phi \wedge G_s, D_s, L_s > == > \ < G_s, D_s^{'}, L_s >} \text{(Inequality I)}$$

In the case the inequality involves a model element, we remove all model elements that are based on the given class from the domain, or any of its subclasses, to maintain consistency with the equality operator.

$$\frac{\begin{array}{c}\mathit{ModelElement2} \lhd \mathit{ModelElement1} \wedge \quad \forall y(y = x \implies \\ D_s^{'}(y) = D_s(y) \setminus \{z : z = \mathit{ModelElement1}(\Phi = P)\}) \wedge \\ \forall y(y \neq x \implies D_s^{'}(y) = D_s(y)) \wedge \\ \{\forall z : (z = \mathit{ModelElement1}(\Phi = P)) \implies \\ (z = \text{instanceOf } \mathit{ModelClass1})\}\end{array}}{\begin{array}{c}< x = \mathit{ModelElement1}(\Phi = P) \wedge \quad G_s, D_s, L_s > \quad == > \\ < G_s, D_s^{'}, L_s >\end{array}} \text{(Inequality II)}$$

Rule Inequality III is very similar, but involves types instead of model elements.

$$\frac{\begin{array}{c}\forall y(y = x \implies D_s^{'}(y) = D_s(y) \setminus \{z : z = \mathit{type}(\Phi = P)\}) \wedge \\ \forall y(y \neq x \implies D_s^{'}(y) = D_s(y))\end{array}}{\begin{array}{c}< x \neq \mathit{type}(\Phi = P) \wedge G_s, D_s, L_s > \quad == > \\ < G_s, D_s^{'}, L_s >\end{array}} \text{(Inequality III)}$$

In other cases, equality may be too strong for what we want to express. In that case, the widening operators $<$ and $>$ can be useful. The constraint handling rules for the widening operator are given below. The basic elimination of widening operators is implemented in rules Widen I and Widen II. Essentially, we restrict the domain of the variable by using the widening rules defined in the type system. Because the widening rules can be defined only for types and model elements, these rules can never be applied if we try to widen any other kind of value, which is the desired semantics of the language. This implements the $<$ operator, the $>$ operator is implemented by the final rule, Widen III. This operator is implemented by translating it to a $<$ operator by switching the arguments.

$$\frac{\begin{array}{c}\forall y(y = x \implies D^{'}_s(y) = \{z|z \in D_s(y) \land \\ z < \phi\}) \land \forall y(y \neq x \implies D^{'}_s(y) = D_s(y))\end{array}}{< x < \phi \land G_s, D_s, L_s > \qquad ==> \qquad < G_s, D^{'}_s, L_s >} \text{ (Widen I)}$$

$$\frac{\begin{array}{c}\forall y(y = x \implies D^{'}_s(y) = \{z|z \in D_s(y) \land \\ \phi < z\}) \land \forall y(y \neq x \implies D^{'}_s(y) = D_s(y))\end{array}}{< \phi < x \land G_s, D_s, L_s > \qquad ==> \qquad < G_s, D^{'}_s, L_s >} \text{ (Widen II)}$$

$$< x > y \land G_s, D_s, L_s > ==> \; < y < x \land G_s, D_s, L_s > \text{ (Widen III)}$$

The next operator we define is the membership operator. The membership operator is used to represent that a variable can have one of several values. This is implemented by creating a new domain from elements that are part of both the old domain and the set of values.

$$\frac{\begin{array}{c}\forall y(y = x \implies D^{'}_s(y) = D_s(y) \cap \Psi) \land \\ \forall y(y \neq x \implies D^{'}_s(y) = D_s(y))\end{array}}{< x \epsilon \Psi \land G_s, D_s, L_s > ==> \; < G_s, D^{'}_s, L_s >} \text{ (Set Inclusion)}$$

Constructs: Loop constraints

A more complex construct is the loop constraint. Loop constraints are used to define constraints for a list of values all at once. Lists are constructed using two operators. The first, [], creates an empty list, and the second, $[Y|Z]$, creates a new list based on the original list Z by adding the value Y to the front. Semantics-wise, we use these list constructors to define these sets of constraints by peeling of values from the list to create the actual instances of the constraints. In the rule, this is represented by the $C[X = Y]$ notation, which means that all instances of X in C are replaced by Y. Note that the size of Y is equal to the size of A, which in turn is equal to the size of X, and X only contains fresh variables. Once the list is empty, the loop constraint represents no constraints anymore and can be removed. Note however, that the loop constraint also constrains the value of the lists that are iterated over. In order to keep this connection, we add extra constraints to ensure that the list of all values remains part of the constraint set.

$$\frac{}{\begin{array}{l}< \mathrm{for}(X = A)^* : C \wedge A = [Y|Z] \wedge G_s, D_s, L_s > \quad\quad ==> \\ < C[X = Y] \wedge A = [Y|Z] \wedge \mathrm{for}(X = Z)^* : C \wedge G_s, D_s, L_s >\end{array}} \text{(For loop I)}$$

$$< \mathrm{for}(X = [])^* : C \wedge G_s, D_s, L_s > ==> \ < G_s, D_s, L_s > \text{(For loop II)}$$

Constructs: Other constraints

The next set of rules implements the various kinds of expressions in EMF-TL. The first four rules, Addition, Subtraction, Multiplication, Division, define the semantics of mathematical operators and are mainly listed for completeness. As soon as both arguments of the operation are instantiated, we use the usual mathematical semantics of these operations to solve the constraints.

$$\frac{\nu_1 + \nu_2 = \nu_3}{< x = \nu_1 + \nu_2 \wedge G_s, D_s, L_s > ==> \ < x = \nu_3 \wedge G_s, D_s, L_s >} \text{(Addition)}$$

$$\frac{\nu_1 - \nu_2 = \nu_3}{< x = \nu_1 - \nu_2 \wedge G_s, D_s, L_s > ==> \ < x = \nu_3 \wedge G_s, D_s, L_s >} \text{(Subtraction)}$$

$$\frac{\nu_1 * \nu_2 = \nu_3}{< x = \nu_1 * \nu_2 \wedge G_s, D_s, L_s > ==> \ < x = \nu_3 \wedge G_s, D_s, L_s >} \text{(Multiplication)}$$

$$\frac{\nu_1 / \nu_2 = \nu_3}{< x = \nu_1 / \nu_2 \wedge G_s, D_s, L_s > ==> \ < x = \nu_3 \wedge G_s, D_s, L_s >} \text{(Division)}$$

The next operations all deal with handling lists. These operations are slightly more complicated in our semantics, because lists are built by list constructors, specifically the constructor for empty lists, [], and the concatenation operator, |, and thus by definition involve nested expressions. This means the initial decomposition of the expressions splits the arguments from the operators. Thus we have to look at two constraints in one rule to define the semantics. Additionally, for some operations, we use the functions defined in Table 4.4 instead of specifying the semantics in detail here, to avoid unnecessary complications. An example of this is the first rule, defining the semantics of LENGTH. As one might expect, this operator is used to get the length of the list, and we use a predefined operator to extract that. The next two operators, FLATTEN and PAIRS, are defined similarly. The FLATTEN operator reduces the nesting from lists, until all elements are part of one top-level list. The PAIRS operator is used to deal with constraints on consecutive elements in a list, for example in situations where the elements represent a sequence of operations where the output of one serves as input for the next. The PAIRS operator addresses this by creating a list of tuples containing an element and its successor, which can then be traversed as normal.

$$\frac{\mathrm{length}\, Y = \nu}{< x = \mathrm{length}\, y \wedge y = Y \wedge G_s, D_s, L_s > ==> \ < x = \nu \wedge G_s, D_s, L_s >} \text{(Length)}$$

$$\frac{\mathrm{flatten}\, y = z}{< x = \mathrm{flatten}\, y \wedge G_s, D_s, L_s > ==> \ < x = z \wedge G_s, D_s, L_s >} \text{(Flatten)}$$

$$\frac{\mathrm{pairs}\, y = z}{< x = \mathrm{pairs}\, y \wedge G_s, D_s, L_s > ==> \ < x = z \wedge G_s, D_s, L_s >} \text{(Pairs)}$$

The other operations, `first` and `last` are simpler, and defined here directly. Because of the way lists are defined here, the `first` operator is very straightforward, because the first element of a list is directly available. The `last` operator is more difficult, because we have to traverse the list to get to its final element. Once we have reached a list with only one element, we know it must be the last element and we have our answer. Note that the patterns listed are intentionally

length	$x =$	case x
		$[y\|z]$: $1 + \text{length}\, z$
		$[]$: 0
		endcase

flatten	$x =$	case x
		$[y\|z]$: if atom y then($[y] \|$ flatten z) else(flatten $y \|$ flatten z) endif
		$[]$: $[]$
		endcase

pairs	$x =$	case x $[y\|[z\|v]]$: $([(y, z)] \| \text{pairs}[z\|v])$ $[y\|[]]$: $[]$ $[]; []$ endcase

Table 4.1: EMF-TL semantics functions

incomplete, because any case where there is no valid first or last element means the computation cannot continue, and thus there is no need to define a rule.

$$\begin{array}{c} < x = \text{first}\, y \wedge y = [p|q] \wedge G_s, D_s, L_s > == > \\ < x = p \wedge G_s, D_s, L_s > \end{array} \qquad (\textsc{First})$$

$$\begin{array}{c} < x = \text{last}\, y \wedge y = [p|q] \wedge G_s, D_s, L_s > == > \\ < x = \text{last}\, y \wedge y = q \wedge G_s, D_s, L_s > \end{array} \qquad (\textsc{Last I})$$

$$\begin{array}{c} < x = \text{last}\, y \wedge y = [q] \wedge G_s, D_s, L_s > == > \\ < x = q \wedge G_s, D_s, L_s > \end{array} \qquad (\textsc{Last II})$$

The last type of expression is the projection. Projections are crucial in EMF-TL, because they are the way type information from other elements is accessed. The main rule in that context is Projection I. In that rule, the variables function is used to find the constraint variable which contains the information desired in the system. Recall that each model element is represented by a set of variables, one for each attribute that can be computed. That means that the combination of an element and an attribute always uniquely correspond to zero or one constraint variables. The variables function is created during constraint generation, and returns the constraint variable that corresponds to a given attribute for an element. If there is no such variable, the function returns value ε instead. Once we have found the constraint variable containing the current value of the attribute, we can merge the domains of the constraint variable with that of the result variable. Because we know the two variables must end up with equal values, we can restrict the domain to only the values that occur in both original domains.

The second projection rule deals with extracting information from model elements. In contrast with the first projection rule, this rule applies only if there is no computed information for this attribute. This is represented in the rule by the condition that there must be no constraint variable corresponding to this element-attribute combination, or in other words, the result of applying the variables function to that. Additionally, the model element must have an attribute with that name. If this is the case, the value of the attribute is extracted from the element and used as answer for the constraint.

$$\frac{\begin{array}{c}\text{variables}(a,\phi) = z \wedge \qquad \forall y(y = x \implies D^{'}_{s}(y) = D_s(z)) \wedge \\ \forall y(y \neq x \implies D^{'}_{s}(y) = D_s(y))\end{array}}{< x = a.\phi \wedge G_s, D_s, L_s > == > \qquad < G_s, D^{'}_{s}, L_s >} \ (\textsc{Projection I}) \ .$$

$$\frac{\text{variables}(y,\phi) = \varepsilon}{\begin{array}{c}< x = y.\phi \wedge y = \mathit{ModelElement1}(\phi = p, \Psi = Q) \wedge G_s, D_s, L_s > \\ == > \\ < x = p \wedge y = \mathit{ModelElement1}(\phi = p, \Psi = Q) \wedge G_s, D_s, L_s >\end{array}} \ (\textsc{Projection II}) \ .$$

Example

Example 11. Recall Example 10. In that example, we discussed how constraints relate to models. In this example, we will discuss constraint handling rules are applied to reach solutions. Suppose we have reached the following state during a constraint computation:

$$\begin{array}{ccc} < & \text{rule}(v_1, v_2) & \wedge \\ & \text{rule}(v_3, v_4) & \wedge \\ & \text{rule}(v_5, v_6), & \\ & \{v_1 \epsilon D_u, v_2 \epsilon D_u, v_3 \epsilon D_u, v_4 D_u, v_5 \epsilon D_u, v_6 \epsilon D_u\}, & \{\} > \end{array}$$

In this state, variables v_1 through v_6 use D_u as the domain of all EMF elements and the constraints are rule(v_1, v_2), rule(v_3, v_4) and rule(v_5, v_6). There are no local variables yet, so that set is empty. We can apply the constraint handling rule defined in Example 10 to the first of these constraints. This means that we remove a match of the left-hand side of the rule, in this case the rule(v_1, v_2) constraint, and replace it by the constraints in the right-hand side of the rule. These constraints represent the conditions that have to be met for this equality expression to be correctly typed. In natural language, the main condition is that the types of the two subexpressions are equal, which is what will be tested during the computation.

This results in a new state:

$$\begin{array}{lcr}
< & o = \text{equals} & \wedge \\
 & l = \text{StringLiteral}(\text{value} = \text{“}\texttt{True}\text{”}) & \wedge \\
 & r = \text{StringLiteral}(\text{value} = \text{“}\texttt{False}\text{”}) & \wedge \\
 & v_1 = i_{equality} & \wedge \\
 & o\,\text{in}\{\text{EQUALS}, \text{DIFFERS}\} & \wedge \\
 & x_1 = r.\text{type} & \wedge \\
 & x_2 = l.\text{type} & \wedge \\
 & x_1 = x_2 & \wedge \\
 & t = \text{slcotype}\{\{\text{type} = \text{BOOLEAN}\}\} & \wedge \\
 & v_2 = t & \wedge \\
 & \text{rule}(v_3, v_4) & \wedge \\
 & \text{rule}(v_5, v_6), & \\
 & \{v_1 \epsilon D_u, v_2 \epsilon D_u, v_3 \epsilon D_u, v_4 \epsilon D_u, v_5 \epsilon D_u, v_6 \epsilon D_u\}, & \\
 & \{o \epsilon D_u, l \epsilon D_u, r \epsilon D_u, x_1 \epsilon D_u, x_2 \epsilon D_u, t \epsilon D_u\} > &
\end{array}$$

In particular, note that the rule application creates a number of local variables, which are added to the corresponding set, while the set of global variables does not grow. Now we have created a number of new constraints, we can apply other rules to remove them. For example, we can apply the rule for set inclusion, Set Inclusion, to this state, to eliminate $o\,\text{in}\{\text{EQUALS}, \text{DIFFERS}\}$. If we do this, the variable o gets the domain $\{\text{EQUALS}, \text{DIFFERS}\}$. We can continue by also eliminating the equality $o = \text{EQUALS}$ with rule Equality I. This further restricts the domain, so it now consists of only the value `equals`. We get the following state:

$$\begin{array}{lcr}
< & l = \text{StringLiteral}(\text{value} = \text{“}\texttt{True}\text{”}) & \wedge \\
 & r = \text{StringLiteral}(\text{value} = \text{“}\texttt{False}\text{”}) & \wedge \\
 & v_1 = i_{equality} & \wedge \\
 & x_1 = r.\text{type} & \wedge \\
 & x_2 = l.\text{type} & \wedge \\
 & x_1 = x_2 & \wedge \\
 & t = \text{slcotype}\{\{\text{type} = \text{BOOLEAN}\}\} & \wedge \\
 & v_2 = t \wedge \text{rule}(v_3, v_4) & \wedge \\
 & \text{rule}(v_5, v_6) & \\
 & \{v_1 \epsilon D_u, v_2 \epsilon D_u, v_3 \epsilon D_u, v_4 \epsilon D_u, v_5 \epsilon D_u, v_6 \epsilon D_u\}, & \\
 & \{o \in \{\text{EQUALS}\}, l \epsilon D_u, r \epsilon D_u, x_1 \epsilon D_u, x_2 \epsilon D_u, t \epsilon D_u\} > &
\end{array}$$

A possible next step is to eliminate the projection operations, i.e. the expressions r.type and l.type. To do that, we first need to eliminate the other two `rule` constraints, to discover the constraints on the projected variables. Because neither the model element in l nor that in r contain information for the type attribute, it must be computed information, and we need to apply rule Projection I. In the context of the rule, the variables r and l correspond to y, and the property type corresponds to ϕ. The function variables will then identify the constraint variables that are used to store the computed types for the two elements. Once more rules have been applied and values for these constraint variables have been found, we insert those values into the other equations. For the purpose of this example, we will not discuss that part of the computation process in detail. Instead, we assume the values of the type attributes of l and r are both $\text{slcotype}\{\{\text{type} = \textsc{String}\}\}$, resulting in the following state:

$$
\begin{array}{lcr}
< & l = \text{StringLiteral}(\text{value} = \text{“}\texttt{True}\text{”}) & \wedge \\
 & r = \text{StringLiteral}(\text{value} = \text{“}\texttt{False}\text{”}) & \wedge \\
 & v_1 = i_{equality} & \wedge \\
 & x_1 = \text{slcotype}\{\{\text{type} = \textsc{String}\}\} & \wedge \\
 & x_2 = \text{slcotype}\{\{\text{type} = \textsc{String}\}\} & \wedge \\
 & x_1 = x_2 & \wedge \\
 & t = \text{slcotype}\{\{\text{type} = \textsc{Boolean}\}\} & \wedge \\
 & v_2 = t & \\
\multicolumn{3}{c}{\{v_1 \epsilon D_u, v_2 \epsilon D_u, v_3 \epsilon \{\text{“StringLiteral”}\}, v_4 \epsilon \{\text{slcotype}\{\{\text{type} = \textsc{String}\}\}\},} \\
\multicolumn{3}{c}{v_5 \epsilon \{\text{“StringLiteral”}\}, v_6 \epsilon \{\text{slcotype}\{\{\text{type} = \textsc{String}\}\}\}\},} \\
\multicolumn{3}{c}{\{o \in \{\textsc{equals}\}, l \epsilon D_u, r \epsilon D_u, x_1 \epsilon \{\text{slcotype}\{\{\text{type} = \textsc{String}\}\}\},} \\
\multicolumn{3}{c}{x_2 \epsilon \{\text{slcotype}\{\{\text{type} = \textsc{String}\}\}\}, t \epsilon D_u\} >}
\end{array}
$$

Now, we have only equalities left. By substituting and applying the equality rules, we can eliminate all constraints, until the goal store becomes empty and `true` is the only remaining constraint. The corresponding state is:

$$
\begin{array}{lc}
< & \text{true}, \\
\multicolumn{2}{c}{\{v_1 \epsilon \{i_{equality}\}, v_2 \epsilon \{\text{slcotype}\{\{\text{type} = \textsc{Boolean}\}\}\},} \\
\multicolumn{2}{c}{v_3 \epsilon \{\text{“StringLiteral”}\}, v_4 \epsilon \{\text{slcotype}\{\{\text{type} = \textsc{String}\}\}\},} \\
\multicolumn{2}{c}{v_5 \epsilon \{\text{“StringLiteral”}\}, v_6 \epsilon \{\text{slcotype}\{\{\text{type} = \textsc{String}\}\}\}\},} \\
\multicolumn{2}{c}{\{o \epsilon \{\text{equals}\}, l \epsilon \{\text{StringLiteral}(\text{value} = \text{“}\texttt{True}\text{”})\},} \\
\multicolumn{2}{c}{r \epsilon \{\text{StringLiteral}(\text{value} = \text{“}\texttt{False}\text{”})\}, x_1 \epsilon \{\text{slcotype}\{\{\text{type} = \textsc{String}\}\}\},} \\
\multicolumn{2}{c}{x_2 \epsilon \{\text{slcotype}\{\{\text{type} = \textsc{String}\}\}\}, t \epsilon \{\text{slcotype}\{\{\text{type} = \textsc{Boolean}\}\}\}\} >}
\end{array}
$$

Because all required constraints have been satisfied, this is a valid final state of the type checker, representing a successful type computation.

This is the only final state, so those values are the solution of the constraint program in this example. However, it will not always be the case that the constraint program has only one final state. In that case, we want to select a single final state that represent the best possible type values according to the strategy defined in the type system. We do this by defining a partial order on solutions, as shown below:

$$V_1 < V_2 \,\text{if}(\forall n : V_1[n] \leq V_2[n]) \wedge (\exists n : V_1[n] < V_2[n])$$

Essentially, we state that one solution, V_1 is “smaller" than another, V_2 if all elements of that solution individually are at least as good as the corresponding elements in the other solution. We use the notation $V_1[n]$ to indicate the nth element of V_1, and $V_2[n]$ for V_2. Additionally, at least one element in V_1 has to be strictly better than its counterpart in V_2. How good elements are in relation to each other is defined by the user in the `strategy` part of the type system. In order to simplify this definition, we use a transitive closure to reduce the number of rules required to create a useable strategy. As an example, consider a binary expression with two number literals. Assuming the number literals can be considered members of several numeric types, like natural, integer or real, there are multiple ways in which this expression can be typed, and thus multiple results. By comparing the results on all elements, we can find the best solution, where all expressions are correctly typed with minimal types.

Once we have ordered the solutions, we look at the minimal solution or solutions. If there are multiple minimal solutions, or if there are multiple solutions and no applicable strategies, we have discovered an ambiguity, and should report that as an error. If there is exactly one minimal solution with respect to our ordering, we have found the results of the type system for the input model. These results consist of both the identifiers of the rules that were used, and the values computed for the features.

Correctness

Now that we have defined how to construct a constraint program and compute a solution, we return to the computation structure we described in Section 4.4. Recall that we want to show that the constraint solution exists if and only if the input model is correctly typed according to the type system. Note that, in our formalism, a model is correctly typed if, for every element that can have a rule applied, a rule must be applied in a way that satisfies all conditions. This is implemented in our semantics by creating an instance of a rule predicate for each element that can have a rule applied. By construction, if a rule predicate is successfully eliminated, that means a rule has been applied. This means a state where all rule predicates have been removed can only be reached if all elements can have rules applied, which means a correct constraint solution exists only if the input model is correctly typed.

Type computation starts with a finite number of model elements, because by definition all EMF models are finite in size. This means we start with a finite number of constraints. Because each constraint handling rule removes at least one constraint and introduces a finite number of new, simpler, constraints, there will always be a finite number of constraints left to consider. In particular, each rule removes at least one element, and creates a limited number of simpler elements, depending on the rule applied. Note that this relation is well-founded, which means elements can not be replaced by simpler ones indefinitely. Also, the rules are defined such that, once an element has been removed, it cannot return. Thus, we know that no rule can be applied infinitely often. This means all constraint computations will at some point reach a state where no further rules can be applied. In such a state, if there are any variables that have domains but are not yet fully instantiated, the solver should insert new equalities based on the domain, potentially creating new possibilities for rules to be applied. At some point, a state will be reached where all variables are instantiated, such that either all constraints have been eliminated, or a number of constraints remain that cannot be eliminated.

In the first case, where all constraints have been eliminated, this means a rule has been applied to each model element, and all conditions have been satisfied. This means the model was correct according to the type system, and the state represents a candidate solution. We assume an ideal constraint solver, which implies that the constraint solver explores all possible combinations of rule applications, and thus it will always find all such states that exist. This means that if there is a solution to the constraint program, our solver will find it. It also implies that if the model is correctly typed, our system will find all sets of values that corresponds to correct type assignments. If the constraint solver is not ideal, there might be values that are not explored, and the constraint solver might not find a solution even if one exists, or may not find the optimal solution. The chance of this depends on how far the constraint solver is from an ideal one. In practice, we found no case yet where the existing ECL^{i}PS^e implementation found a non-ideal solution.

Overall, we can be sure our constraint program results in one or more type assignments if and only if the input model is correctly typed. Each of these assignments represents a way in which we can create a correct output model. However, we want to create only one output model. One solution is to construct a general output model that can represent all solutions. However, because our output model should conform to a specific metamodel, we cannot be sure this is possible. Instead, we choose to select one solution from the result set based on the strategy defined in the type system. Recall that in the strategy section of the type system, the language designer can describe which types are considered better than others. However, a solution does not just consist of one type value, but of several. If one of the solutions is better than all others as defined in Section 4.4, it is the optimal choice and we can use it. If there are multiple solutions that are both better and worse than each other in some aspects, the result is ambiguous. In that case, no output model can be generated. In terms of correctness, this means our system does not find a solution for ambiguous cases, even though they exist. We find this acceptable, because two output models that are very similar from a type system perspective, may have very different semantics from a user perspective. Thus, we consider it better not to risk creating an erroneous output model than to create a model despite our uncertainty.

If the end state contains unresolved predicates, this means for this combination of rules not all conditions can be met. If all end states of a constraint program are of this type, that means there is no combination of rule applications that does not have contradictions with either the model or other rules, which means the model is incorrectly typed. Similarly, if a model is incorrectly typed, this means that for every possible combination of rule applications, there is at least one condition that cannot be satisfied. This condition can be of various types, but in all cases, the condition will translate to one or more constraints. Because the model is incorrect, we know that at least one constraint cannot be eliminated because the preconditions for that are not met. For example, if there are two equalities that equate the same variables with different values, one cannot be eliminated because the corresponding position in the variable store is already filled. Thus, if the model is incorrect, the constraint program will have no valid end states.

Overview

Now that we have defined the semantics of the constraint part, we have to look at how it fits in the whole process. We will do this in the form of a proof sketch. We will try to show that our type systems are decidable, i.e. we can always consistently determine whether a given model element can have given type and the type system will always return the same result when repeatedly applied to the same input model. First, we observe that the semantics of the transformation depend strongly on the results of the constraint program, because we need to know which rules were applied to which elements in order to create correct equivalents in the output model. Fortunately, the constraint part is itself independent of the transformation, and the constraint part is guaranteed to terminate, which means we can safely assume the two parts are executed sequentially. In particular, if the constraint solving step finds no solution, we can be sure that we will not be able to generate a correct output model, so it makes no sense to apply the transformation semantics. For the following discussion, we will therefore assume that correct solutions exists.

Like for the constraint part, we know that the input model of the transformation is finite by construction, and so is the number of rules in the type system. We know that for every element that requires it, we have exactly one set of corresponding constraint results that uniquely identify of a type system rule that can be applied to it. Recall that each rule describes a finite number of elements which must be created in the output model. This means that the number of elements created in the transformation to apply type system rules is always finite. For elements that have no relevant rules, the corresponding element in the output model is by construction uniquely defined. Overall, this means that the construction of the output model will always terminate. In addition, because the rules are deterministic and the results of the constraint phase are consistent, the created model is also consistent.

Overall, we conclude that the semantics of type systems in our language are decidable, consistent and terminating. Note that this does not mean that all type systems are sensible. It is certainly possible to write a type system with no solutions for some or even all models. It does mean that the “errors" in the models will be identified consistently in a finite amount of time.

4.5 Implementation of EMF-TL semantics

Once we have type system specifications in EMF-TL, we want to create type checkers that implement them. Recall that in Section 4.4, we defined the type checker as a two-phase process. The two phases are the *constraint* phase and the *model generation* phase. The first phase is itself divided into two steps, *constraint generation* and *constraint solving*. In our implementation we follow this structure directly. We have implemented a constraint generator as a combination of an ATL [64] transformation and an Xtend [38] template. The transformation is itself generated by a *type checker generator*, based on the type system specification, while the template is reused between different type checkers. After the constraints program has been generated, we use the ECL^{i}PS^e constraint solver [6] to compute its solutions. The reason for choosing ECL^{i}PS^e rather than other constraint systems is the support for so-called structures, data types that are similar to `records` and `structs` in traditional programming languages, and have a name and a set of fields. Structures allow us to represent compound types in a convenient and direct way, making the prototype easier to debug and simplifying the conversions to and from the constraint solver. While ECL^{i}PS^e is not an ideal constraint solver, meaning that it does not guarantee all possible values will be explored, as assumed by our semantics, we found it sufficiently powerful to solve practical instances.

Once the constraint solver finishes, the constraint phase is completed, and we move on to the model generation process. This phase is implemented as a Java model transformation, which we refer to as the *result merger*. If the input model is correctly typed, it combines the untyped input model and the constraint results into a typed model. In contrast, if the input model contained type errors, it will collect and create a list of the errors and output that instead. The reason we choose different approaches is that the constraint generator transformation has to create an extensive constraint model, that contains the constraints describing the relations between model mments and their computed values, while the result merger mainly inserts elements directly based on the constraints results. In fact, the result merger needs no knowledge about the metamodel beyond what is provided in the type system description. Thus, it was possible to build a generic component rather than a generator for the second transformation, but not for the first.

The *type checker generator*, which creates the type-system specific constraints generator, is itself based on two ATL transformations. These transformations, inspired by an example by ATLflow tool [114], generalize the ATL refinement mode. The first transformation takes an EMF metamodel and creates a model of a generic transformation that creates an explicit copy of each element occurring in a model described by the metamodel. The second ATL transformation is then used as a template to create an ATL transformation that implements the generic transformation. The basic template copies all elements from one model to another. Based on the type system specification, the generated rules are adapted by adding extra target elements to a generated rule. This means extra model elements are created during the copying process. Those extra elements are used to represent constraints in the constraint program. This approach does mean that we get an ATL transformation with a number of rules equal to the number of concrete classes in the input metamodel, which can be quite large, but in our experience this does not significantly impact performance. In our current implementation, we use an EMFtext parser to convert the type system specification from text into suitable input for the type checker generator. We then generate the constraint generator and insert it into the the overall process as the second step. How we implement the various constructs used in the semantics of EMF-TL in our chosen constraint solver will be discussed in Section 4.5.

The second transformation of the type checker is the result merger. Its main function is to copy all elements from the successfully typed model to the final model, while inserting all computed values produced by the constraint generator into the appropriate attributes. Note that these two models will likely have different metamodels, so we cannot simply insert the new elements into the first model. We assume however, that the two metamodels are closely related, in that all elements not mentioned in the type system can be copied to a corresponding element in the new model. The second transformation is implemented using the Java interface to Ecore [94], the data format used for EMF models. This interface provides functionality not only to access and create model elements, but also to access the corresponding metamodel class and its properties. By using this reflection, we can identify which attributes occur in both source and target version of the element, and copy them where possible. Additionally, if we encounter an element that has a constraint variable, we use the results of the constraint solving step to create a new model element and to assign the proper values to its attributes. Once this process is complete, the result is a model based on the target metamodel that contains all defined type information. Note that the result merger can only do this if there are no type errors in the model. If there are errors, in general we cannot be sure we can create a consistent model, because we may miss required information for some elements.

Constraint generation

The constraint generation process works by analyzing each rule for each element in turn. Remember that each EMF-TL rule is restricted to a specific metaclass of model elements. If an element fits none of these restrictions, we know the type system does not cover it and can safely ignore it. If an element does have the right metaclass for at least one rule, we call it a *typed element*. We know that, for a model to be correctly typed, at least one rule must be applied to each typed element. We represent the application of the rule in the constraint program by a set of domain-less constraint variables that should contain an identifier of the rule that was used for this element and the other values that are computed for it. In contrast with the theoretical semantics, instead of initially assigning a universal domain to the variables, we simple use the possibility ECL^{i}PS^e offers not to assign it a domain at all. We specifically do not assign domains to the variables before type computation, because these domains are often structurally complex and infinite in size. Due to their complexity, no constraint solver we know supports them directly, and their infinite size, combined with a non-linear structure, makes it not possible to map them elegantly to domains that constraints solvers do support, like natural numbers or sets thereof. For example, consider a type representing containers with a set type of elements and a set size. There is an infinite number of possible container sizes, and, because the containers can be nested, an infinite number of element types. This means that it is not possible to map all possible combinations onto separate integers. The nesting also makes it impossible to encode this type as a set of natural numbers, because we cannot distinguish between the encoding of the size and the encoding of the element type. A consequence of this is that we cannot apply many constraint solving techniques to these variables, but must rely on propagation to discover their values. In practice, we find this causes no problems.

To ensure the identifier in the constraint variables is valid, we add a set of conditional constraints that each represent the application of a type rule as described in Section 4.4. A difference with the theoretical semantics is that for each variable that is not used in a given rule, a domain is used consisting of only an explicit error value. Other constraints can refer to these variables to determine if their conditions are compatible with the constraints already in place.

In summary, we create a set of constraint handling rules for each rule for each element of the appropriate metaclass. To show these constraint handling rules implement the applicability of each rule on the corresponding elements correctly, we have to show that the conditions of the constraint handling rules are equivalent to the preconditions of the corresponding type rule. The relation between basic preconditions of rules and constraint conditions is shown in Figure 4.7. We use the decomposition described in Section 4.4 to implement the other conditions listed in Figure 4.4 by a combination of the conditions below and fresh local variables.

In Figure 4.7, the EMF-TL precondition is shown on the left, the corresponding ECL^{i}PS^e constraint on the right. For the first kind of condition, we used the `compare` predicate. This predicate holds if either the ECL^{i}PS^e =, which implements unification without occurs-check, holds, or if the inheritance equivalence defined in Section 4.4 holds. Note again that in EMF-TL, a value is considered equal only to itself. Also, because our goal is to find values for the variables that fit all constraints, we consider unification a valid implementation for the EMF-TL equality condition. For the second kind of condition, we have to show that the predicate `widen` corresponds to the partial order defined in the specification. The `widen` predicate is added to the constraint program during constraint generation and computes the transitive closure of a predicate encoding the `<` relations defined in the widening section of the type system, as described in Section 4.3. The conditions of the defined widenings are translated in the same way as rule conditions. If the defined partial order is not circu-

1	*var1* = *var2*	`compare`(*var1 var2*)
2	*var1* $\Leftarrow$ *var2*	`widen`(*var1*,*var2*)
3	*var1* >= *var2*	`widen`(*var2*,*var1*)
4	*var1* `in` *var2*	`member`(*var1*,*var2*)
5	*var1* = *type*((*attribute* = *var*)*)	*var1* = *type*((*attribute* = *var*)*)
6	*var1* = *MM1::ModelElement1* ((*attribute* = *var*)*)	*var1* = *ModelElement1* ((*attribute* = *var*)*)
7	*var1* = *var2.attribute*	`hastype`(*var2*,$I_{attribute}$,*var1*)
8	`for` (*var* = *val*)* `:` (*cond*)*	((`foreach`(*var*, *val*))* `do` (*constraint*)*)

Figure 4.7: Basic conditions

lar, finiteness of the input ensures termination of `widen`. The partial order is not checked for circularity during constraint generation, because that cannot be conveniently done in ATL, but the `widen` predicate checks that the same type is not visited twice, thus preventing infinite loops that would prevent the type checker from terminating. The third kind of condition is implemented using the set membership predicate `member` that is part of the standard library of ECL^{i}PS^e . This predicate holds if *var2* is a collection and *var1* is an element of that collection. In particular, this instantiates *var1* if it has not been instantiated yet. Note that because we do not use domains, we use a combination of instantiation and backtracking instead. This is less efficient than the definition using domains given in the formal semantics, but the resulting solutions should be identical. The fourth and fifth type of condition unify the value of a variable with a specific type structure. Again, this is already part of the standard behavior ECL^{i}PS^e uses for =. The sixth kind of condition, referencing computed attributes for other elements, is implemented using the `hastype` predicate. This predicate computes which constraint variable corresponds the indicated attribute of the given model element. It then uses = to constrain *var1* to be unifiable with that constraint variable. The final kind of condition is the for-condition. This kind of condition was inspired by, and is implemented with, an ECL^{i}PS^e loop construct. This construct applies the constraints in the body of the loop to each set of variables generated by the generator. We only use one form of generator here, namely one that iterates over several lists in parallel. Note that, due to the finite size of input models, we can be sure that the loops will always terminate.

In order to translate conditions, we also have to translate expressions. In Figure 4.8, we show EMF-TL expressions on the left, and the corresponding ECL^{i}PS^e expression on the right. The first four lines describe the various basic expressions. These can and are transferred directly to the constraint program. The next two lines describe set and list literals. For simplicity, these are both translated as ECL^{i}PS^e list constructs. The next line, line 7, describe the EMF-

1	*number*	*number*
2	*"string"*	*"string"*
3	`OclUndefined`	`"oclUndefined"`
4	*var*	*var*
5	$\texttt{Set\{}\boldsymbol{exp}^*\texttt{\}}$	exp^*
6	$\texttt{List\{}\boldsymbol{exp}^*\texttt{\}}$	exp^*
7	$exp1\ (+ \mid - \mid * \mid \&)\ exp2$	$(+ \mid - \mid * \mid \text{concat})(exp1, exp2, res)$
8	$(\text{length} \mid \text{first} \mid \text{last} \mid \text{flatten} \mid \text{pairs})\ exp$	$(\text{length} \mid \text{first} \mid \text{last} \mid \text{flatten} \mid \text{pairs})\ exp$
9	$type((attribute = exp)^*)^*)$	$type(attribute = exp)^*$
10	*MetaModel1::ModelElement1* $((attribute = exp)^*)$	*MetaModel1::ModelElement1* $\{(attribute : exp)^*\}$
11	*exp1.attribute*	getconstraint(*exp1*,*attribute*)

Figure 4.8: Expressions

TL binary operators and their ECL^{i}PS^e equivalent. A peculiarity of ECL^{i}PS^e is that it does not directly feature mathematical operators in an infix style, but only in prefix form. Line 8 describes the unary operators, which are all translated to ECL^{i}PS^e functions directly. The next two lines describe how types and model elements are translated into ECL^{i}PS^e structures. Finally, the last line describes how the projection operator is implemented by translating it into a call to the getconstraint function.

Summarizing the previous discussion, we argue that we have created a set of constraint variables and corresponding constraints for each rule and model element, and that the conditions of the rules are correctly implemented in the constraints. This ensures that the set of (type) assignments that satisfy all the rules of the type system is equal to the set of solutions of the constraint program. However, we do not want a set of solutions, we want our preferred solution according to the defined strategy. In order to find this solution, once the constraints are constructed, we add two additional predicates to the program: `strategy` and `cost`. The `strategy` predicate holds if the two parameters are directly linked by the defined strategy. The `cost` predicate uses the `strategy` predicate to assign a cost to each type. All types that are minimal according to the strategy, are assigned cost 1. If a type is not minimal, it gets a cost of 10 times the cost of the type directly below it in the strategy. This way, we can quantify the quality of a solution by computing the cost of all types computed for all elements, which in turn makes it possible to apply standard optimization algorithms. The best solution according to the type order is the solution with the lowest cost among all solutions. This solution can be found using the branch-and-bound algorithm provided by ECL^{i}PS^e .

We then rely on the third step, *constraint solving* to actually compute our solution. The correctness of this step depends on the correctness of the ECL^{i}PS^e constraint solver and the provided predicates. We know that ECL^{i}PS^e is not an ideal constraint solver, but we do believe that it is strong enough to cover all practical, or even possible, program instances that are generated by our type checker. However, because ECL^{i}PS^e is such a complex system, we consider the exact definition of the power required, and the proof that ECL^{i}PS^e can provide it, beyond the scope of this thesis.

Finally, we have to show that the result merger creates correct output models based on the output of the constraint solver. This transformation is conceptually not that complicated, but because the implementation is very generic, it is quite large. Thus, any proof of its correctness is likely to be very large and tedious. Thus, we choose not to prove this at this point. We will provide the source code of the result merger for inspection as an on-line appendix [78]. Instead, we currently use testing as a means to check the correctness of our implementation. By applying the generated type checker to test cases, we can compare its results to the expected results.

4.6 Related work

In the previous Chapters 3 and 3.9, we used Modular Structural Operational Semantics [85] (MSOS) to specify a type system. In that work, we focused on specification as opposed to implementation. In the current chapter, we use our own DSL, called EMF-TL, inspired by MSOS, to specify type systems. By using our own DSL, we are able to extend our specification with direct links to metamodels and explicit strategy elements, that are not present in basic MSOS. We focus on using generated transformations and constraint solving to compute the solutions to a typing problem. Constraints are useful to represent situations where information is available about what type a given expression is going to have, but not enough to select the right type assignment immediately. By defining a constraint, we can represent this knowledge without requiring us to make a choice before we want to. We will use this to represent, for example, the specific properties of widening numeric values in the CIF language in Chapter 5.2. This property of constraint-based type checking has hitherto mostly been studied in the context of complex and mostly implicit type systems, like those that occur in functional programming languages [2, 92]. Despite the existing work on generic type checking frameworks [102], we do not know of any type checker generators based on constraint solving. Most DSLs will not use the full power that the constraint system offers, but it does allow us to define type systems with few restrictions. This may come at a price in performance, but without alternative implementations of the same type systems to compare our implementations with, this is hard to assess.

One system that does use a limited form of constraints is XTypeS [14]. XTypeS is a system designed to define type systems for validating EMF models created by XText parsers. A type system in XTypeS is defined using rules, consisting of a so-called typing judgement and optionally some conditions. The judgement describes the result of the rule, the knowledge gained when all conditions hold. The rules define relations between elements. Note that, in contrast to our approach, XTypeS is focused on validation of the correctness of the types, not on type inference. This means that the types computed by the type system are not inserted into the model, but discarded. While this ensures that the model stays consistent with the same metamodel, any discarded information cannot be easily used by other tools. Another difference is that there is no specific notion of 'type' in XTypeS. Also, there can be only one rule per element type for each judgment, unlike our approach, where there can be arbitrarily many rules for each type of element. That means that there is no difference between defining one type to be the subtype of another or assigning a type to an element, since both are but relations between elements. In particular, the authors of XTypeS use this to implement unification in the rules, thus allowing simple constraints. This differs from our approach, where we use an external solver to handle constraints. Once a type system has been specified using XTypeS, a Java implementation can be generated that applies the rules to given models. In order to facilitate validation, the system allows rules that relate elements to the special element "OK", indicating a correctly-typed element. If this is done, the generated code provides functions to check if a given model element is correctly typed. If there are any errors, the generated type checker throws an exception, which can be analyzed to pinpoint the source of the error. In comparison with our approach, XTypeS is more closely integrated with existing editors, but it is less flexible and less useful as part of a tool chain.

Another, more basic system aimed at providing type checkers for EMF models that uses a constraints-like approach is XText/TS [112]. This system aims at expressions in EMF models, and consists of a Java API that provides a number of templates of type constraints. For example, a type system designer can specify that for a certain feature of a certain class of elements, only elements with certain types are allowed. By instantiating these constraints with references to the metamodel of the model to be typed, a type system can be built up. The instantiation is done by invoking API functions with appropriate arguments. In addition to the provided templates, a *CustomTypeChecker* can be implemented in Java to provide more customized rules, but this reduces the advantages of using the API. In particular, the API lacks templates dealing with features that can contain multiple elements. This alone would mean that implementing the type system for CIF expressions that have a varying number of subexpressions, like list constants, in XText/TS would require extensive customization. Additionally, like XTypeS, XText/TS only validates models and does not insert type information into the model.

In addition to EMF-based approaches, some other language workbenches have dedicated mechanisms to define type checkers. An example of this is the MPS [63] system that is based on JetBrains. Type checkers are implemented in MPS by using *language aspects* that implement inference rules using the *type system language*. The type system is specified using rules that create constraints on types when applied to elements that meet their condition. Once a rule is applied, so-called *typeOf* expressions can be used to trigger the application of rules to subelements. Once all constraints have been collected,the type system engine will then compute types that meet all constraints. In contrast to our approach, MPS updates the input model with the type information, but it is not possible to transform an element based on the type computed. Additionally, MPS uses a less powerful constraint system then we do, which makes some rules harder and less elegant to express.

In his book ‘Types and Programming Languages’ [88], Benjamin Pierce selected Structured Operational Semantics [89] (SOS) as the preferred method of defining semantics, and uses a similar formalism, natural deduction, to specify type systems. In many ways, this formalism is quite similar to the one we use, except it directly relates elements to types instead of to typed elements. Compared to our approach, this makes the rules more compact, but removes the possibility of transforming the element to a different class during the transformation process. For example, a generic reference element can be specialized into a specific reference type depending on what is referenced using our system, but not using the system used in ‘Types and Programming Languages’. Moreover, like type system specifications that are SOS based, environments are explicit, even if they are not relevant for the current rule.

During the writing of his book, Pierce together with Michael Levin constructed a tool to verify that the type system in the book were correct. This tool is called *TinkerType* [76]. In contrast with our tool, which is focussed on type systems tailored to the needs of a specific DSL, TinkerType is focused on combining so-called *typing features* into coherent type systems. This is done using the TinkerType assembler, a program that takes typing features from two repositories and creates either a typeset version of the type system or an ML implementation. One consequence of this is that all typing features appear to assume the same structure of the language structures to be type checked, in a kind of implicit metamodel. This greatly reduces the practical usefulness of the generated type checkers. Additionally, in contrast to our type systems, TinkerType typing features actually contain a significant amount of direct ML code, making it harder to construct code generators for other languages. Finally, the website given in the paper is unfortunately no longer directly online, which suggests to us that TinkerType is no longer developed or supported.

Another well-known practical method to define semantics of languages, and static semantics like type systems in particular, is attribute grammars. Attribute grammars describe type computation by defining it as an attributes for relevant nodes, together with expressions to compute it based on other attributes. In order to generate a type checker using attribute grammars, we need an attribute grammar system that creates an implementation based on the attributes’ specification. Several well-known systems in this area are UU-AG [103], JastAdd [56] and Silver [108]. As far as we are aware, only JastAdd can be applied to EMF models, using the work of Bürger et al. [26]. Their approach is quite complex, requiring several interventions in the normal workflow of JastAdd to function. Additionally, because their approach is more generic, it offers no explicit support for more type system specific features, like strategies for disambiguation. There are attribute grammar-based systems specific for typing, but they offer no support for EMF [7, 51].

4.7 Conclusions

In this chapter, we have introduced the EMF-TL type system language. In this language, transformations can be specified that can add type information to models, converting an untyped version to a typed version. We have defined the syntax and semantics for this language, and discussed an implementation and its correctness. The language was created in response to our experiences with MSOS, aiming to allow more freedom in defining type rules, and to create an easier interface with other language tools. We addressed the first target by integrating a number of constraint-solving concepts into our language. These concepts allow the designer to delay choices and to keep options open as long as possible. The second target was addressed by using the EMF framework as a foundation for our language. The EMF framework is based around models, and specifically around metamodels that model the structure of other models. The metamodels are used both to create model infrastructure, and to allow tools to share models easily.

In EMF-TL, this results in a type system specification consisting of a number of type rules, that each describe for a class of model elements a way to add type information to it, and when that way is valid. By combining rules for various classes, and even different rules for the same class, we can cover a wide range of typing scenarios. In order type type a model, we try to find a consistent set of rules such that exactly one rule is applied to each element, and all conditions are met. If we can find such a set, we know the model is correctly typed and can construct an output model that contains the computed type information. If not, we can identify the elements where no valid rule can be found and use that information to report errors to the user.

We created a prototype implementation for EMF-TL, and applied it to the modeling language SLCO in this chapter for demonstration purposes. In the next chapter, we will explore how well EMF-TL meets the requirements outlined in Section 2.4, and how well it serves as an answer to RQ 1.3.

Chapter 5

Case studies

5.1 Introduction

In this chapter, we will describe a number of more extensive case studies. Based on our SLR, we selected four languages as subjects of our experiments. A primary factor in the selection of these languages was the presence of experts that were available to be consulted on the structure of the language and the correctness of the defined type system. The four languages also all have pre-existing type systems, to ensure the design of the type system was not affected by any limitations in EMF-TL. Instead, we would adapt EMF-TL to accommodate problematic constructs. All languages we selected are strongly typed, because we found most statically-typed DSLs are, as described in Section 2.2. In each case study, we studied the publicly available documentation of the language, and used that to create a type system specification. We then generated the corresponding type checker, and applied it to test cases. In the next subsections, we will first introduce the four languages in more detail, and describe our reasons for using them in our case studies. We will then describe the general procedure used to implement the case studies, followed by the case studies themselves. Each case study ends with an evaluation section, which discusses our results. Finally, the final section of this chapter contains the overall evaluation, were we compare the case study results from the different languages and draw our end conclusions.

CIF

The first language we selected is CIF, the successor to the Chi 2.0 language we studied in Chapters 3 and 3.9. Like Chi 2.0, CIF is a domain-specific language describing behavior of hybrid systems, i.e. systems combining state machines that define discrete behavior and differential equations that define continuous behavior. Unlike Chi 2.0, CIF was designed from the beginning around EMF. It was also designed to be an interchange format, containing a wider array of constructs to ensure that all required semantics could be described in CIF models. Like in Chi 2.0, a CIF model consists of combination of discrete and continuous elements. The continuous elements are modeled using differential equation systems, and are mostly used to model physical processes occurring in the modeled system. The discrete elements are modeled using state machines and are primarily used to represent the controllers used to keep the system operating within the desired parameters. Together, these controllers form a model that can be used to examine, for example, how various implementations affect the overall functioning of a complete system containing both electronic and mechanical components. This case study can be found in Section 5.2. By studying CIF, we create an opportunity to compare the MSOS and EMF-TL specifications of the same type system.

WebDSL

The next language we choose was WebDSL, an object-oriented language for web design. WebDSL was created in 2007 by Eelco Visser [111]. It is different from HTML, a major DSL used for constructing web pages out of graphical elements, in that it explicitly models data and operations that link together pages in addition to visual representations of those pages. A complete website in WebDSL is referred to as an *application* or *app*. An app can define *entities* that define the structure of data, *functions* that operate on this data and *pages* that define how entities are displayed to the user. In terms of our SLR, specifically Section 2.2, WebDSL is a statically and strongly typed language. It contains objects with inheritance, type variables and overloading, but not type inference. We choose to study a object-oriented language because object-orientation is an influential paradigm in general purpose programming languages, and we believe it can offer useful abstractions for a number of domains. Object-orientation also poses specific type system challenges, and we wanted to make sure EMF-TL can meet those challenges. In fact, this case study led to a change in EMF-TL, namely the introduction of the *in* construct, intended to express implicit references found in this language.

mCRL2

The third language we choose was the mCRL2 language. mCRL2 [52] is a specification language based on the Algebra of Communicating Processes(ACP) [13] extended with abstract data types. mCRL2 is based on the earlier μCRL language [53] and intended to create system specifications that can serve as a basis for property verification through model-checking. To that end, a number of tools exist that can convert system specifications into state space representations, which can then be analyzed to determine if they posses desired or undesired properties. In terms of our SLR from Section 2.2, mCRL2 is statically and strongly typed. It is not object-oriented, but uses algebraic data types. It features overloading and type variables, and a limited amount of type inference, as defined in Section 2.2.

Unlike CIF, mCRL2 is not an EMF based language. Thus, in order to apply our generated type checker to real models, we created our own metamodel for mCRL2, based on the grammar given for mCRL2 in the documentation. For the sake of convenience, we also created an Xtext parser that can convert textual example models into instances of the metamodel. Unfortunately, because the parser produced by Xtext [39] is not as powerful as the one is in the actual mCRL2 toolset, some constructs cannot be added to the grammar without creating ambiguities that the parser cannot deal with. In particular, this affects the bag literal expression and the `where` clause, which can be used to structure expressions by giving names to subexpressions that occur multiple times. Fortunately, the constructs affected are not present in many example models, so we can still use our parser to access the models easily. Additionally, we feel the affected constructs are not particularly interesting from a type system perspective, so there is little lost by not including them. Note that we could have used other methods of creating models to allow these constructs to be used anyway, but because we want to use existing mCRL2 models to test our type checker, we choose to put our focus elsewhere.

POOSL

Finally, Section 5.5 describes our experiences with another specification language, this time an object-oriented one called POOSL. POOSL is a domain-specific language for system modeling developed at the Eindhoven University of Technology in 1997 and primarily used for performance modeling. POOSL is part of the Software/Hardware Engineering (SHE) [93] design method for complex reactive systems. POOSL is aimed at large communicating systems, and it features much stronger layering that the other three case study languages. It is also an example of a language with a mostly undocumented type system. Based on our SLR results in Section 2.2, we observe that this is likely the case for many DSLs, making this an important case to consider.

Case Study Procedure

In all cases, the case studies were carried out by the author of this thesis. This creates a threat to validity, because the useability of the language by its intended users can obviously not be derived from the experiences of the language designer alone. However, based on the prototype nature of the available tooling, it was decided that the increased effort that would be required to a significant number of other users to the case studies would restrict the number of case studies that could be conducted in the time available too much. Another threat to validity is that we could not compare our generated type checkers to existing implementations directly, because these either did not exist or were not compatible with the EMF framework. We felt this issue was not insurmountable, because an user of the type system will also usually not base his knowledge of the type system of a language on an analysis of the type checker, but on the available documentation. Thus, the documentation should provide a complete and consistent picture of the type system, allowing us to base our type system specification on it.

5.2 Case Study: CIF expressions

As mentioned in the case study introduction in Section 5, the basic expression constructs in CIF are similar to those found in Chi 2.0, and also similar to those found in many GPLs. There are numerical values, strings and booleans, with their corresponding types. Unlike Chi 2.0, we do not need the additional types `cnat` and `cint` here, because we implement the behavior they are used for in a different way. There are also various kinds of containers, including lists, arrays and two-dimensional matrices, and we have implemented type rules for all of these. Expressions can be combined using operators or used as arguments for function calls. As in most programming languages, the user can define variables and constants of these types, and refer to these as needed. In the case of variables, the hybrid nature of CIF shows itself in the presence of so-called *continuous* and *algebraic* variables. In contrast with *discrete* variables, of which the value only changes when explicitly updated, values of continuous or algebraic variables can change depending on the passing of time. If, for example, we want to model the volume of a liquid contained in a storage vessel, we can use a continuous variable to represent the fact liquid enters or leaves the vessel as a flow, instead of updating the volume discretely at set intervals.

In addition to variables and constants, users of CIF can also define their own functions. These functions can then be called, assigned to variables or passed as arguments to other functions. To keep things simple, however, user-defined functions cannot be overloaded or generic. Overloading and type variables are present, but reserved for predefined library functions. Because CIF users are commonly mechanical engineers with little programming experience, the designers of the language felt that allowing overloading would result in a lot of errors, for little benefit. By restricting advanced features to predefined libraries, the designers can provide, for example, a generic function that determines the length of lists with arbitrary types of elements, without exposing this complexity to the user. Another peculiarity of CIF is that type widening is allowed for constants, but not allowed for variables, as described in Section 3.4.

CIF Typesystem

The first three parts of the EMF-TL specification of CIF are shown in Listing 5.1. The `imports` part specifies that the metamodels of the language in question are `http://ucif.tue.nl/ucif-1.0.0` and `http://cif.tue.nl/cif-1.0.0`. The first is the source metamodel (the "u" stands for "untyped"), the second is the target metamodel. The current implementation does not require this order, but we find this is a useful convention. The `typesystem` part defines what types exist in this fragment of CIF, and how they are represented in the target model. In our example the type system has fourteen types, representing natural, integer and real numbers, booleans, strings, matrices, arrays, vectors, lists, sets, functions, distributions, dictionaries and tuples. The first five are basic types, with no further properties, while the latter nine are composite types with parameters. The `type order` part of the type specification defines a partial order of these types. For the base types, `nat` is defined to be a subtype of `int`, and `int` is a subtype of `real`. For the composite types, one instance of `array` is defined to be a subtype of another instance of `array` if their dimension parameters match and the type of the elements of the first is a subtype of the type of elements of the second. The subtype relation for lists is defined similarly, based on comparison of the types of the elements. Remember that for the sake of simplicity, the subtype relation is assumed to be reflexive, so every type is a subtype of itself, and transitive, so `nat` is also a subtype of `real`, etc.

Listing 5.1: CIF type system fragment

```
imports

 http://ucif.tue.nl/ucif-1.0.0;
 http://cif.tue.nl/cif-2.1.1;

start

    ucif::Specification

typesystem

bool = cif::types::BoolType;
real =cif::types::RealType;
int = cif::types::IntType;
nat = cif::types::NatType;
string = cif::types::StringType;

matrix(x, y, e) = cif::types::MatrixType
  (rowDimension,columnDimension,elementType);
array(d, e) =
  cif::types::ArrayType(dimension,elementType);
vector(d, e) =
  cif::types::VectorType(dimension,elementType);
list(e) = cif::types::ListType(elementType);
cifset(e) = cif::types::SetType(elementType);
function(p,r) =
  cif::types::FunctionType(parameterTypes,
  returnType);
distributiontype(r) =
  cif::types::DistributionType(resultType);
dictionary(k,v) =
  cif::types::DictionaryType(keyType,valueType);
tuple(f) = cif::types::TupleType(fields);
```

```

widening

nat< int
int< real

list(e=$e1) < list(e=$e2) if $e1 < $e2
array(d=$d1,e=$e1) < array(d=$d2, e=$e2) if $d1 =
    $d2 , $e1 < $e2
```

Listing 5.2: CIF type system strategy

```
strategy
nat< int
int< real
list(e=$e1) < list(e=$e2) if $e1 < $e2
array(d=$d1,e=$e1) < array(d=$d2, e=$e2) if $d1 =
    $d2 , $e1 < $e2
```

The fourth part of the type specification, entitled `strategy` and shown in Listing 5.2, indicates priorities among different valid type assignments. It is defined in the same way as the type order, and will usually have a significant similarity to it. For instance, one can type the constant 1 as either being `nat`, `int` or `real`. This is an example where a simple and generally applicable rule leads to ambiguity. In some cases, this ambiguity is an error and should be reported as such by the type checker. For instance, given two potential types $(\texttt{nat}, \texttt{int}) \to \texttt{int}$ and $(\texttt{nat}, \texttt{int}) \to \texttt{nat}$ for the function reference f, the expression $f(1, 2)$ can be associated with `nat` or `int` depending on which function type is chosen by the type checker. In some cases, however, the type checker should be capable of selecting one of the solutions as the “most appropriate” one. For the constant 1 one can expect the type checker to prefer the smallest possible subtype determined according to `type order`, i.e., `nat`. Sometimes, however, the smallest type according to the `type order` is not optimal based on other considerations: for function types we may want to consider $\texttt{int} \to \texttt{nat}$ as a subtype of $\texttt{nat} \to \texttt{nat}$, because any function that can be applied to integers can also be applied to naturals, but still prefer the latter type over the former when selecting the expression type, for example because it can be implemented more efficiently. For example, if we know that the type of the parameter is `int`, we need to consider negative numbers, but if the type checker has derived the smaller type `nat`, we know these will not occur.

Listing 5.3: CIF Number rule

```
from ucif::expressions::Number
to cif::expressions::Number(type=$t)
where $t< real{{}}
```

Listing 5.4: CIF RealNumber rule

```
from ucif::expressions::RealNumber
to cif::expressions::RealNumber(type=$t)
where $t = real{{}}
```

Listing 5.5: CIF FunctionCallExpression rule

```
from ucif::expressions::FunctionCallExpression( function = $function, arguments = $arguments)
with $functiontype, $parametertypes
to cif::expressions::FunctionCallExpression( type = $returntype)
where $functiontype = $function.type,
      $functiontype = function{{p = $parametertypes, r = $returntype}},
      for $pt in $parametertypes, $a in $arguments : $pt = $a.type
```

The fifth and final part of the specification, called `rules`, shown in Listings 5.3, 5.4 and 5.5, describes how various model elements are typed. The `from` clause determines the applicability of the rule. For instance, the rules in Listings 5.3 and 5.4 are applicable only to `ucif::expressions::Number` elements, in case of the first rule, and `ucif::expressions::RealNumber` elements, in case of the second rule. These rules constrain the types of the elements to subtypes of `real` and only `real` respectively. Note that any type is considered a subtype of itself. The `with` clause lists local variables used in the rule. The `to` clause introduces the target model element that will be created when the rule is applied. Finally, the `when` clause specifies conditions that should be satisfied by the model element and its related elements. For instance, the `when` clause of the rule in Listing 5.5 requires that the type of the subexpression `function` is a function, and that the values of the `type` attribute of the `arguments` subexpressions match the types defined for the parameters of the function. Note that we could also have chosen to require only that the type of the parameters are larger than the types computed for the attributes, but the designers of CIF prefer the stricter version shown here. In this formulation, if a `ucif::expressions::Number` element is provided as an argument for a function with a parameter of type `real`, the `ucif::expressions::Number` element ends up with type `real`[1]. This is because the value `real` meets both the constraint imposed by the rule for `ucif::expressions::Number` elements, and the constraint imposed by the rule for `ucif::expressions::FunctionCallExpression` elements on its arguments. If we would provide a model element that cannot be of type `real` as argument, this rule cannot be applied and if there are no other rules that can be applied, type checking fails.

[1] For disambiguation purposes, we have to add `{{}}` to the type name, but this does not affect the type itself.

Evaluation

As CIF is the successor language of Chi 2, and in particular the expressions of CIF are directly based on Chi 2.0 expressions, we can compare the specifications of the two type systems. One direct conclusion that can be drawn is that, because EMF-TL rules are more verbose than MSOS rules, the CIF type system specification is significantly longer than the Chi 2.0 one. This can be easily seen by comparing the sizes of two example rules, as shown in Figure 5.1 and Listing 5.6. The MSOS implementation applies only to one operator, while the EMF-TL implementation applies to several, but this is a result of the different structure of the parse tree used, and does not significantly affect the comparison. As can be seen from the examples, it is especially the more detailed descriptions of the model elements involved, including references to metamodels, that make the EMF-TL rule larger. The two specifications define similar type systems, but the EMF-TL specification is more flexible, because it allows additional kinds of conditions to be used in the specification and because it is tied to a metamodel instead of a parse tree format, allowing easier interaction with other tools, like for example visual CIF editors. An example of the additional flexibility can also be seen in the rules in Figure 5.1 and Listing 5.6. In the MSOS version, we have to use a maximum operator here, creating the need for separate types to indicate if widening is or is not possible. In the EMF-TL version, we can use the equality operator, allowing the decision to widen or not to be made elsewhere, in a position where more information is available.

The metamodel also allows us to check if all model elements referenced are used correctly, reducing the number of potential errors in the specification. In the MSDF specification, any misspelling in references to the input parse tree can only be detected at run time, while the EMF-TL editor can detect this statically. In terms of performance, the EMF-TL-based type checker is slightly slower than the MSOS-based one, taking 50% longer for similar size models, but we believe there is further room for optimization, especially in the way strategies are evaluated on various constraint terms. If the expected performance increase can be achieved, the impact of using EMF-TL instead of MSOS should be negligible.

Listing 5.6: CIF binary operator rule

```
from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $u
to cif::expressions::BinaryExpression(type=$t)
where $o in set{
  ucif::expressions::BinaryOperators::GreaterEqual,
  ucif::expressions::BinaryOperators::GreaterThan,
  ucif::expressions::BinaryOperators::LessEqual,
  ucif::expressions::BinaryOperators::LessThan},
    $u = $l.type,
    $u = $r.type,
     $u < real{{}},
     $t = bool{{}}
```

```
E1 =={---}=> T1, E2 =={---}=> T2,
max(T1,T2) = T3, max(T3,real) = real
--------------------------
less(E1,E2) : Exp =={---}=> bool .
```

Figure 5.1: MSDF rule for a binary operator.

5.3 Case Study: WebDSL

As mentioned in the introduction in Section 5, WebDSL is an object-oriented language, that makes use of constructs called entities. A WebDSL entity is similar to a class in EMF or in an object-oriented programming language, and an example definition of a `User` entity is shown in Listing 5.10. Every entity has a name, and can have a *superentity*, *properties* and *functions*. A property in WebDSL is similar to a structural feature in EMF, and can contain a value or a (containment) reference in the same way as in EMF. In the example, we can see four value properties: `username`, `password`, `name` and `isAdmin`. Two of those have type `String`, one has type `Bool` and one has type `Secret`. Secret as a type is fundamentally similar to String, but using this type indicates the value of the property is special and should be protected. The example entity also has two references, `manager` and `employees`, both of which refer to other values of `User` type. Note that in contrast to EMF, a WebDSL property does not have multiplicity. If a property should contain multiple values, this is done by using a collection, like a set or a bag, as used for the `employees` property. There are, however, a number of other annotations that can be added to properties, like assertions that restrict the values that the property can have, or indicate that the property can be used as an identifier for an entity.

Listing 5.7: WebDSL entity example

```
entity User {
username :: String (id)
password :: Secret
name :: String
manager -> User
employees -> Set<User>
isAdmin :: Bool
}
```

A WebDSL entity function describes behavior related to a specific entity. A common use for functions is to test whether an instance of an entity meets certain criteria. For example, if the entity represents a user, a function might test if the user has reached a certain age based on their birth date. Note that WebDSL functions, despite the name, are not always free of side effects. A good example is constructor functions. Each entity has an implicit constructor function, and if these were side-effect free, we would expect that invoking the same constructor multiple times would return the same object every time. However, if one of the properties has been defined as an identity, the same identifier will never be used twice, or in other words, invoking the constructor has the side effect of removing one identifier from the set of available options. As a main consequence, all entities created by that constructor will be different from each other.

In order to display the data in entities to users of the web site, and to allow entities to be created and edited, we need the other main type of WebDSL construct, the *page*. An example of a page definition is shown in Listing 5.8. Pages in WebDSL are in some ways similar to procedures in imperative programming languages. Each page definition has a name, in the example `editUser`, and optionally some parameters, in the example one, namely `u`, and can be invoked to create an HTML page based on the actual argument values, which is then displayed to the user. The structure of a page is defined by using elements that are similar to HTML tags. Examples of elements are headers, forms and links, as shown in Listing 5.8. The page in the example actually creates a form that can be used to edit some fields of an `User` entity, and a button to save the new values. There are also some control flow statements that can be used in pages, for example loops to display all elements of a collection, but they are not used in the example shown. In order to enhance reuse, we can also define *templates* that can be invoked from pages to recreate specific groups of elements, for example a common footer, for a number of pages. A template consists of the same kind of elements as a page, but no new page is created when it is invoked. Instead, the elements are added to an existing page. Unlike pages, templates can be overloaded. This is useful to create specialized representations for values of different types. For example, suppose one wants to display a list with several kinds of items in it. By using an overloaded template, one can use a simple template call and still get an appropriate output for each item.

Listing 5.8: WebDSL page example

```
define page editUser (u : User) {
  title { "Edit User: " output(u.name) }
  section {
    header { "Edit User: " output(u.name) }
    form {
      par { "Name: " input(u.name) }
      par { "Password: " input(u.password) }
      par { action("Save Changes", saveUser()) }
    }
    action saveUser() {
      u.persist(); return viewUser(u);
    }
    navigate(home()) { "return to home page" }
  }
}
```

If entities could only be used to display information, WebDSL would not add much to pure HTML. However, we can also define *actions* in WebDSL that can be linked to submission of forms or other triggers. Using actions, we can save information to a database, update the information on a page or create new pages. Actions are defined using imperative statements, including control flow like loops and switches, but primarily assignments and creation of new entity instances.

WebDSL Type System

Because our SLCO case study covered most basic expressions, we choose this next case study based on more advanced type system features. In particular, we selected WebDSL because it contains both object orientation and overloading, two characteristics that also featured in our SLR in Section 2.2. Both represent specific challenges to our type system language, because they involve interaction between scoping and typing, which are two separate concepts in our approach. The rule examples in this chapter are all extracts from the actual WebDSL EMF-TL specification, which can be found in Appendix C. The examples can be located within the complete specification using the line numbers.

Object-orientation

As the name of the concept "object-orientation" implies, WebDSL, or any other object-oriented language, is built around the concept of objects. An object is a compound structure that consists of values and references. What kind of values and references an object can have is defined by a so-called *class*. In WebDSL, classes work similarly to those found in many object-oriented languages, like for example EMF, with the exception that they are referred to as *entities*. For the purpose of this general discussion, we will use the more generically used name *class*. To recap the description of WebDSL entities given in Section 5.3, classes define *features*, which can contain values, and can also define behavior, as *functions* or *procedures*, that is usually directly tied to the object.[2] Classes are related to each other through a concept called *inheritance*. Each class can have one or more parent classes, from which it *inherits* features and behavior.

In object-oriented languages, objects are created, used, updated and destroyed. In order to do this, we need to resolve references to classes and features in order to determine what types are involved. For example, if we define a class representing a *car*, we may want to refer to specific parts of the car, like its engine or one of the doors. Usually, we will want to refer to a particular part in the context of a particular car. To enable this, in addition to plain references, object-oriented languages use *qualified* references. Continuing our car example,

[2] For a more concrete discussion of classes and features, please refer to Section 4.2.

an instance of a qualified reference in natural language would be the phrase "The roof of the blue car". In this phrase, the identifier "roof" is qualified by the phrase "the blue car", restricting it from a general concept to a specific reference. Recall that classes allow programmers to define data with internal structures, creating the need to refer to specific parts of specific elements. Instead of one name, a qualified reference essentially nests references to allow the programmer to define a reference to a component of an entity. In each step, the outer reference targets an element that defines a scope, and the inner reference targets an element in that scope. In many ways, this is similar to ordinary references, but the first complication is that feature names can be reused between classes. In our example, we could create several classes to represent different kinds of cars, that all have a feature named engine. This is usually not a problem, because a feature can only be accessed for a specific object, which has a specific class, and each class can have only one feature with a given name. This does mean that we need to know the class that defines the object before we can resolve the feature reference. A second complication is that due to inheritance, the feature may not be structurally part of the class that is directly referenced by the object. Instead, the feature could be one that is inherited from an ancestor class. In our example, we could make the engine feature part of a general *car* class, with several other classes implementing more specific kinds of cars inheriting this feature from it. This inheritance chain could itself contain type errors, for example ambiguous superclass references or classes overriding features from ancestors in an illegal way.

In our two-phase type checking process, as described in Section 4.3, we resolve qualified references by explicitly using the ability of the scoping phase to return sets of potential targets instead of single elements, in combination with the nested nature of qualified references. The result is that we treat the innermost reference as an ordinary reference, but each nesting checks whether the choice made in the inner reference is consistent with the choice made in the outer one. The rule that specifies this is shown in Listing 5.9. As in the rules for SLCO, lines 133, 134 and 135 define the class the rule applies to and the class that will be used in the target model. In this case, the class representing quantified references in WebDSL is called `FieldAccess`. The relevant properties of the field access are the name of the target property and the expression that represents the base object. In line 136-138, the rule extracts the type of the base expression from the `resultSort` property, then the actual entity from the type, and finally the features of the entity from the `allBody` property. The rule then selects a field from the potential candidates identified by scoping in line 139, checks that the property indeed has the correct name, and finally verifies that the property is part of the correct entity. If this is the case, then the rule can be fully applied and the `field` attribute of the resulting element will be set to refer to the computed field.

Listing 5.9: WebDSL field access rule

```
133    from textualWebDSL::FieldAccess(links = $links,
         base = $exp, position = $pos, field = $name)
```

```
134    with $exptype, $entity, $props, $name
135    to WebDSL::FieldAccess(field = $field,
         resultSort = $t)
136    where $exptype = $exp.resultSort,
137          $entity = $exptype.entity,
138          $props = $entity.allBody,
139          $field in $links,
140          $field = textualWebDSL::Property(sort =
               $t, name = $name),
141          $field in $props
```

Note that the feature `allBody`, referred to in line 138, contains not only the properties directly contained in the entity, but also those inherited from super classes. It is not directly filled by the parser, but a result of type computation. This computation is specified in the rule shown in Listing 5.10 that type checks entity definitions. The rule does not compute an actual type for an entity definition, because an entity is already a type itself. Instead, a type checking rule is used to resolve the superentity reference, and to compute the inherited properties. The specification is actually rather straightforward. The first two lines of the precondition select the target entity and are similar to other reference rules. Line 305 extracts the features, both direct and inherited, of the superentity, and the last line adds the local features to compute the desired set of features, which is stored in `allBody`.

Listing 5.10: WebDSL entity definition rule

```
300
301    from textualWebDSL::Entity(body = $body, links
         = $links, superentity = $s)
302    with $superb
303    to WebDSL::Entity(superentity = $e, allBody =
         $allbody)
304    where $e in $links,
305          $e = WebDSL::Entity(name = $s),
306          $superb = $e.allBody,
```

Overloading

Though strictly speaking unrelated, overloading presents similar challenges to object orientation. Recall that overloading refers to identifiers that are used for multiple similar entities, like multiple functions with similar, but different signatures. For each reference that uses an overloaded identifier, for example as part of a function call, we have to use additional information, like the types of the arguments of the call, to decide which entity is the actual intended target. Like with qualified references, we solve this by designing the scoping to return all possible targets for a reference, and then choose the actual target during typing. The rules involved, however, tend to be more complex than those for objects.

For one, functions tend to have a variable number of arguments. Additionally, instead of a simple test if the target occurs in a certain place, deciding between overloaded targets can involve a more subtle judgment based on several factors. In EMF-TL, the way functions are compared to make this choice is defined using the `strategy` part of the type system, as introduced in Section 4.3. There, the type system designer can define what functions are preferred over other, for example by comparing the types of parameters or the return types.

In the actual type system specification, the function call is represented by two rules, one to handle instances where the function explicitly is linked to an object, and one for instances where a global function, one that is not accompanied by a reference to an object. In Listing 5.11, we show the latter of those two rules. The lines 198-200 describe the source and target elements of the rule. As part of this, we state that we are interested in three properties of the input element, namely `base`, the base expression, `links`, the candidate functions discovered during the scoping step and `arguments`, the expressions that provide the argument values for the function call. The base expression of a function call is the expression that determines the object the function is part of, if one is provided. The more interesting part is in the preconditions of the rule. In line 201, the rule tests if the base expression is indeed empty. In WebDSL, the reference to the function to be called is part of the call itself. Thus, the next line ensures the function is valid according to the scoping rules, and line 203 checks that it it is indeed a function and extracts the parameters and the return type. The next line extracts a list of parameter types from the list of parameters. This list essentially represents the parameter part of the function signature. In the final line, line 205, the most important work is done. In this loop, we compare the type of each argument to the type of the corresponding parameter, using the widening operator. If all the arguments are compatible, we know the chosen function call can be called here and could be part of the final result.

Listing 5.11: WebDSL function call rule

```
198
199    from textualWebDSL::Call(base = $base, links =
         $links, arguments = $args)
200    with $fparams, $fparamtypes
201    to WebDSL::Call(function = $f, resultSort = $t)
202    where $base = OclUndefined,
203          $f in $links,
204          $f = textualWebDSL::Function(arguments =
               $fparams, returnSort=$t),
205          for $fparam in $fparams, $fparamtype in
               $fparamtypes : $fparam =
               textualWebDSL::FormalArg(sort =
               $fparamtype),
```

The case for calls to functions linked to specific objects is similar, and we will not show it separately here. The only difference is that instead of testing if the base expression is empty, the rule tests that there is an expression. The type of this expression should be an object type, and the function called should be part of that object. That part of the rule is similar to the one discussed in Section 5.3.

As mentioned before, if the rule shown in Listing 5.11 has multiple answers for a certain model element, we need to decide between them according to some metric. In WebDSL, the selection is based on the following two rules. In line 323, the specification states that the left function is preferable to the right one only if they have the same name and if all parameters of the left function are preferred over the corresponding parameter in the right function. In turn, preference between parameters is described in line 324, and simply corresponds to preference between the types of the parameters. As an example, consider two functions, both called "double", one with a parameter of sort `Integer` and the other with a parameter of sort `Float`. If we compare these two functions using the preference rule, we first check if their names are equal, which is the case. We then check for each parameter, in this case only one, how they are related. This triggers the second preference rule. If we assume that elsewhere in the specification, a preference rule has been defined that indicates that `Integer < Float`, the second rule will indicate that `Integer` parameters are preferred over `Float` parameters. As a result, the function with the `Integer` sort parameter will be preferred over the function with the `Float` sort parameter.

Listing 5.12: WebDSL function strategy rule

```
323    simpletype(sort = $s1) < simpletype(sort = $s2)
         if $s1 = "Integer", $s2 = "Float"
324  WebDSL::Function(name = $n1, arguments = $args1)
       < WebDSL::Function(name = $n2, arguments =
       $args2)
325        if $n1 = $n2,
326           for $arg1 in $args1,
327           $arg2 in $args2 : $arg1 < $arg2
328  WebDSL::FormalArg(sort = $s1) <
       WebDSL::FormalArg(sort = $s2)
```

Evaluation

Unlike for SLCO and CIF, the tool suite of WebDSL already included a type checker for the language. This type checker is defined by a Stratego/XL [110] program. Because this type checker is implemented in a more mature system than our prototype, it makes little sense to compare them quantitatively performance-wise. Generally speaking, we expect our prototype to be significantly slower, using seconds to type models rather than the tens or hundreds of seconds used by the production type checker. Further optimizations should remedy to a significant extent, however. Another difference is that we have chosen not to focus on the sublanguages of WebDSL, focussing instead on the core features of the main language. This automatically makes our specification considerably less complete. On the other hand, we feel the declarative nature of our specifications makes them both easier to understand and easier to maintain.

5.4 Case Study: mCRL2

Every mCRL2 specification consists of abstract data types and algebraic processes, with possibly added functions and equations. Abstract data types are used to model the data that flows through the system as processes communicate. Each abstract data type, known as a *sort*, is essentially a non-empty set of data elements. Data elements are defined using constructors. Each constructor has a name, optionally some parameters and belongs to a specific sort. The simplest constructors have no parameters, and thus add at most one element to the sort each. More advanced constructors add new data elements based on smaller data elements. An example of a sort can be seen in Listing 5.13. In this definition, we define a sort for binary trees that contain integer numbers in the leaves. The `leaf` constructor is essentially the base case of the sort, and has one `Int` parameter, representing the value stored in the leaf. The `node` constructor combines two binary trees into one bigger binary tree.

Listing 5.13: mCRL2 sort example

```
sort BinaryTree = struct leaf(value : Int) |
                         node(left : BinaryTree,
                           right : BinaryTree);
```

The other crucial part of a mCRL2 specification are the processes. Two example processes are shown in Listing 5.14, which will be discussed in greater detail later in this section. The processes define the behavior of the system. In mCRL2, a process consists of actions, which are also defined in the specification. Each action can have parameters, that contain the data involved in the action. In Listing 5.14, the actions defined are `call`, `answer`, `exchange`, `done` and `alldone`. Once actions are defined, they can be combined into processes with several operators. The primary operators are the alternative composition, written as `+`, which creates a choice between two options, and sequential composition, written as `.`, which creates an order between two actions. These two operators are independent of data, meaning for example the choice among the alternatives cannot be influenced by the values of the parameters of the action.

Listing 5.14: mCRL2 example

```
map N: Pos;
    Information: Set(Pos);
eqn N = 5;
    Information = { k:Pos | 1 <= k  && k <= N };

act done, all_done;
```

```
 7      call, answer, exchange: Pos # Pos # Set(Pos)
          # Set(Pos);
 8
 9  proc Person(myid:Pos,myin:Set(Pos)) =
10       sum herid:Pos, herin:Set(Pos) . (
11           ( myid != herid ) -> (
12               ( call(myid,herid,myin,herin)
13                 + answer(herid,myid,herin,myin) ) .
14               Person(myid,myin + herin) ) ) +
15       ( myin == Information ) -> done . delta;
16
17  proc Person_init(id:Pos) = Person(id,{id});
18
19  init allow({exchange,all_done},
20       comm({call|answer -> exchange,
21       done|done|done|done|done-> all_done },
22         Person_init(1) || Person_init(2) ||
23           Person_init(3) ||
23         Person_init(4) || Person_init(5)
24       ));
```

If we do want data to influence behavior, we need to use two further operators, the conditional and the sum operator. The conditional operator selects an action based on the result of an expression, similar to the well-known if-then-else-statement in traditional programming languages, and is shown in lines 11-14 of the example. The `sum` operator, shown in line 10, is an generalization of the alternative operator. Instead of a choice between two options, it defines a choice between instances of the same action that differ in one or more parameters. To be more precise, there is one instance of the action for every data element in a particular sort. Because sorts can contain an infinite number of elements, this can not be replicated by using the ordinary alternative operator. A common case where the sum operator can be useful are actions that represent the reception of a message. If the message contains, for example, a positive number, there are an infinite number of messages that the sender can choose to send. In order to model that we can receive any of them, we can use the sum operator with the sort representing natural numbers to model all possibilities.

Another major construct that can be used in creating processes is recursion. Recall that recursion in general refers to items that contain one or more copies of themselves. In order to keep process specifications of manageable size, mCRL2 uses the concept of process variables. These process variables can not only be referenced in other processes to access the process contained within, but also in the process being assigned itself. This way, processes can contain direct or indirect references to themselves, allowing processes of infinite length to be specified. This may sound strange, but in practice it is a convenient way to specify the behavior of many systems. Like actions, process variables can have parameters that define the data involved. This is useful, for example, to specify a protocol for sending or receiving messages in greater detail, without having to repeat the definition of the protocol in several places in the specification. In Listing 5.14, we can see recursion in process `Person`, defined in lines 9–15 and referenced from inside its own definition in line 14.

The previous constructs all describe a single process operating in isolation. In reality, a system may have multiple components that act largely independently, but do communicate. In that case, we do want to model them together. For this purpose, there is a parallel composition operator, represented by the `||` operator, that enables two processes to act separately. This is often needed to represent independent systems, like separate computers, that do need to interact at some points. Once we have multiple parallel processes, it becomes possible for them to act simultaneously. This is referred to as a multi-action, and an example where this must occur is when two processes communicate in a synchronous manner. In synchronous communication, the message is sent and received at the same time, like for example in a telephone conversation. In mCRL2, this kind of communication can be modeled by using the `comm` operator, used in line 20 of the example in Listing 5.14. With this operator, we can define groups of actions that can be merged together into one communication action when the data parameters are equal. Most commonly, one of the actions will represent the sending of a message, for example someone speaking, another will represent the reception of a message, for example someone listening, and the data parameters represent the contents of the message, for example the text spoken. Note that while the communication operator describes how actions can combine, it does not force these actions to only occur together. This means that, for example, the action to hear something could be taken without anything being said. To model situations where that is not desired, the `allow` operator is used, as shown in line 19 of the example. By applying the allow operator to a process, we can restrict the actions that can be taken to a specific set. If we put only the combined action into the set, the components of the communication cannot be performed in isolation.

Finally, there are three operators that can be used for modeling, but are primarily used to enable reuse or improve analysis of the resulting transition system. These operators are blocking, renaming and hiding. As the name suggests, the `block` operator prevents actions from a given set from occurring. In essence, it is the opposite of the `allow` operator. The `block` operator is most often used to remove some uninteresting behavior from the system, allowing the analysis to focus on more important actions. The `rename` operator allows, as the name suggests, actions to be renamed. This can be useful, for example, when we want to reuse a process in several places, and want to distinguish the instances. The final operator is the `hide` operator. This operator essentially makes all actions from a given set silent. A silent action, conventionally represented using the τ symbol or `tau`, can still be executed, but cannot be observed directly. Proper use of silent actions can dramatically reduce the size of the observable state space of systems, making analysis much easier.

Listing 5.15: mCRL2 Phone Book Function example

```
 1  sort  Name;
 2        PhoneNumber;
 3        PhoneBook = Name -> PhoneNumber;
 4
 5  map   p0: PhoneNumber;
 6        emptybook: PhoneBook;
 7        add_phone: PhoneBook # Name # PhoneNumber ->
            PhoneBook;
 8        del_phone: PhoneBook # Name -> PhoneBook;
 9        find_phone: PhoneBook # Name -> PhoneNumber;
10
11  eqn   emptybook = lambda n: Name . p0;
12
13  var b: PhoneBook;
14      n: Name;
15      p: PhoneNumber;
16  eqn   add_phone(b, n, p) = b[n->p];
17        del_phone(b, n) = b[n->p0];
18        find_phone(b, n) = b(n);
```

In addition to sorts and processes in mCRL2, we can define functions and equations. Like constructors, a function definition is essentially only a signature. Examples of such signatures are shown in Listing 5.15. We first define three sorts for representing a simple phone book, that links names to numbers. In the example, we show two functions, called maps in mCRL2, with no arguments in lines 5 and 6, which are used to create specific constant values, namely the empty phone number and the empty phone book, when invoked. The other functions all take an existing phone book as a parameter, and either search, add or remove phone numbers in it. Unlike constructors, functions have actual behavior, defined in equations. In principle, an equation defines two terms to be equal. In practice, equations are often defined with a complex function term on one side and a smaller result on the other side. For example, if we want to define a function representing addition, we specify one or more equations with an addition on one side and a simpler addition, or even a direct result, on the other side. Ideally, by repeatedly applying equations, we can reduce any addition to a single final result. In the example[3] in Listing 5.15, we show four equations defining the behavior of the declared functions. The first one, in line 11, defines a phone book that contains an empty phone number for every entry. Next, in lines 13–15, we declare a number of variables. These are used in the next three equations, in lines 16–18, which define the behavior of the functions that modify phone books. In the first two, we use the map update operator to either replace a phone number by a new one, in the `add_phone` function, or to replace a phone number by an empty one, in the `del_phone` function. In the

[3] This example is based on `http://www.mcrl2.org/release/user_manual/tutorial/phonebook/index.html`

last, the map access operator is used to extract the number corresponding to the given name from the phone book.

An example[4] of an mCRL2 specification can be seen in Figure 5.14. In this case, the language is used to describe a communication problem, where we want to know the minimum number of actions needed for a number of people to share information. In the beginning, each person has exactly one part of the information. People can share information by calling each other. After each call, the caller and the callee have shared all their information, so they know the same things. The process is complete when all people know all information.

In the model, we first define the information, in lines 1–4. The information is represented as a set with each information item represented as a number, specifically a positive number of the sort `Pos`. In this instance of the problem, there are five pieces of information, so the set contains 5 elements. The next step is to define the actions that are involved, in lines 6 and 7. The first two actions `done` and `all_done` are introduced as indications that the process is done. The first one, `done`, is used by each participant to indicate that he or she has all information. Once all participants have all information, they together take the `all_done` action to indicate the problem is solved. The other actions, `call`, `answer` and `exchange` represent the communication between people. They each have four parameters, with the first two representing the people involved in the call, and the second two representing their knowledge. Note that in both cases, we are not interested in the details of the person or fact involved, but only in their existence as separate entities, thus, numbers are used to represent them.

The next and main part of the specification is the definition of the people. The behavior of each person is defined by a recursive process. In this process, the person gets a choice between communicating with another, as defined in lines 10–14, or stopping, as defined in line 15. The second option has a so-called *guard*, which states that it can only be used once the person has collected all information. Note that a person can still choose to communicate once this guard holds. If the person chooses to stop, the `done` action has to be taken, which can only be done by communicating with all other participants. If all participants take the `done` action, we know that all pieces of information have spread to all people, and the problem is solved. We then use the predefined `delta` action, also referred to as *deadlock*, to indicate that the process is over. If the choice is made to communicate, the person has to decide who to communicate with. This is specified by the `sum` operator in line 10. The sum allows the person to choose any communication partner, and to receive any information item form them. In the next line, line 11, a guard is used to eliminate the possibility of a person calling itself. While such a call would not affect the validity of a solution, it does not result in any progress towards the goal. The next step is to choose between communicating by calling someone, or by being called by someone, using alternate composition operator, written as `+`. This is represented by the choice in lines 12 and 13, between the call action and the answer action. After

[4] This example is based on `http://www.mcrl2.org/release/user_manual/tutorial/gossip/index.html`

communication, the person now knows all information the other person knew, in addition to his own. This is accomplished using the set union operator, which is also written as `+`, an example of operator overloading in mCRL2. So, once the call is compete, the process repeats, but with the person's knowledge updated to a new value.

The next code line, line 17, defines a convenience process that initializes a person with their own piece of information. It is used in the next part, lines 19–24, that defines the actual system. In lines 22 and 23, the five people involved are created as parallel processes. As mentioned before, if defined like that, they can take actions independently. But in this case, it does not make sense to `answer` a call that has not been made, or vice versa. Thus, we need to define the actions as communications, and this is done in lines 20 and 21. In the first of those lines, we define that `call` and `answer` combine to create an `exchange`. Essentially, the `exchange` action defines a complete phone call. In the next line, we define that the action `all_done` consists of five parallel `done` actions. This action serves mainly a modeling use, because when it can be used, we know all five people are done, and everyone has all information. Note, however, that despite having defined what sensible phone calls are, the `call` and `answer` action can still be done separately. This is solved with line 19, with the `allow` operator. By allowing only complete communications, we ensure the separate communication actions, like `call` and `answer`, do not cause any problems.

mCRL2 Type System

The mCRL2 type system is static, strong and explicit. It is not object-oriented, so it does not feature objects or inheritance. It does feature type parameters, and a limited form of type inference. In particular, references to actions and processes can be ambiguous. For instance, in Listing 5.14, we use two atomic actions, `call` and `answer`, to represent a phone conversation in an abstract way. If we wanted to model a phone conversation in greater detail, we could choose to define `call` and `answer` as processes instead, allowing us to define how separate steps together make a conversation. When invoking `call` or `answer` however, like is done in line 12 and 13, it makes no difference whether they are defined as actions or processes: the syntax is identical. Instead, the type system is used to decide for each invocation whether the referenced behavior is defined as an action or as a process. In EMF-TL, this is implemented by a combination of rules that transforms the base reference into an action reference or a process reference depending on the target that is found. This disambiguation is discussed in more detail in Section 5.4. Another important feature of the mCRL2 type system lies in sort aliases. In mCRL2, users can not only use built-in sorts and define new sorts themselves, new names can also be given to existing sorts. These aliases can be used to, for example, create names for complex container types that are easier to use and update. From a type system perspective, this means we have to take into account that the same type might occur in the system under different names. Because mCRL2 semantics define that the fundamental type should be used to determine if a given expression is valid, no matter how it was referenced, this means that we need normalization to discover the underlying type for each sort expression, so we can apply the type system rules based on the correct values. This is discussed in greater detail in Section 5.4. Our full mCRL2 type system formalization can be found in Appendix D.

Element Transformation

In Section 5.3 of the WebDSL case study, we discussed how to handle the concept of overloading in EMF-TL. In WebDSL, overloading represents the possibility to define multiple functions with the same name. This means it cannot be syntactically decided which of the implementations is referenced by a given function invocation, so the invocations have to be disambiguated based on their type signature during type checking. In mCRL2, functions can be overloaded, but there is an additional complication in the form of actions and processes. Recall that actions represent basic units of behavior, and can be combined in processes to define more complex behavior. Both actions and processes have parameters, and invocations of actions and processes can be syntactically very similar. While this flexible syntax makes the language more user-friendly, as a consequence, the parsers cannot decide whether a given invocation refers to an action or a process. This means that in the untyped model, invocations of both actions and processes are represented using the same class of elements. In the typed model, however, we would like to use model elements with direct references to the action or process invoked, and this can be best represented by using separate classes for action invocations and process invocations. We implement this in EMF-TL by using multiple rules that transform model elements based on the computed types.

In the rules, the disambiguation is primarily represented by the first two rules shown in Listing 5.16, in lines 73 to 90. Note that while these two rules share the same source element, they have differing target elements, representing different semantics. Because actions and processes in mCRL2 are almost similar from a typing perspective, the two rules are nearly identical also. In the first line of the conditions, we select the relevant action or process from the environment. In the second line, we test whether we are dealing with an action or a process by using a reference to the `Atom` and `Process`, and if the name is correct, we also extract the parameters from the element. In the third line, we then test if the types of the argument expressions match the type of the parameters of the action or process.

The third rule in Listing 5.16, in lines 92 to 96, deals with some specific semantics for processes. In order to simplify the definition of recursive processes, a shorthand form can be used where parameters can be copied from an invoking process implicitly. As a result, a process can be invoked in its own definition without giving explicit arguments for all its parameters. Unfortunately, due to limitations of the parser we used to implement the textual syntax of mCRL2, we do not fully support this feature, but only the version where all argument values are copied from the parent expression, the remainder being possible future work. In the rule, the first thing to note is the reference to the parent process next to the source element of the rule. This reference gives us both a way to test if the process call is actually contained in a process definition, and not, for example, part of an initialization expression, but also gives us access to the process element, so we can check if the properties of the element match where required. In the conditions, we check if the number of arguments is indeed zero, in the first line. In the next line, we extract the process from the parent process declaration. Finally, we check if the process name matches the name used in the invocation, to ensure the call is really recursive. If these conditions hold, the call represents a valid recursive invocation, and is typed correctly.

Listing 5.16: mCRL2 AtomicAction rules

```
from textualmcrl2::AtomicAction(atomname =
  $name, links = $env, arguments = $args)
with $t
to mcrl2::AtomicAction(atom = $a)
where $a in $env,
      $a = textualmcrl2::Atom(name = $name,
        type = $t),
      for $arg in $args,
          $param in $t :
          $arg.type.sort < $param.sort

from textualmcrl2::AtomicAction(atomname =
  $name, links = $env, arguments = $args)
with $t
to mcrl2::Instance(process = $p)
where $p in $env,
      $p = textualmcrl2::Process
              (name = $name, parameters = $t),
      for $arg in $args,
          $param in $t :
          $arg.type.sort < $param.sort.sort

from textualmcrl2::AtomicAction(atomname =
  $name, links = $env, arguments = $args) in
  $parent: textualmcrl2::ProcessDecl
to mcrl2::Instance(process = $p)
where length $args = 0,
      $parent =
        textualmcrl2::ProcessDecl(process =
        $p),
      $p = textualmcrl2::Process(name=$name)
```

Sort Renaming and Normalization

Another important aspect of the mCRL2 type system is sort renaming. A designer can introduce a new name for any sort, and then refer to the sort using that name throughout the model. In fact, as designed in mCRL2, such a renaming essentially creates an alias, using the new name of the sort is the same as using the sort directly. This contrasts with other languages, like WebDSL, where you cannot define new names for built-in types. During type checking, we have to make sure that two aliases of the same sort are properly treated as equal. In other words, we cannot use the elements in the model defined by the user directly in the type system, but we have to do a so-called *normalization* to compute the actual sort represented.

In our metamodel of mCRL2, we implemented this concept by creating a clear distinction between sorts, represented using the Sort class and its subclasses, and sort expressions, represented by the subclasses of SortExpr. In the type system specification, this means that to relate SortExprs to Sorts, we need a number of rules like those shown in Listings 5.17, 5.18 and 5.19. In the first rule, a BoolSort element is added to a Bool sort expression. This rule is basic in structure, because that is the only sort that this expression can represent. If multiple sorts could be represented by this expression, we would likely need more conditions to determine which is applicable in which situation. The next rule, in Listing 5.18, adds a ListSort element to a List sort expression. Because list sorts in mCRL2 are parameterized with the sort of elements they can contain, we have to take the sort of a child sort expression and add it to the resulting ListSort. The final rules, in Listing 5.19, address sort reference expressions. The first rule handles references to structured sorts, and creates a direct reference to the relevant sort. The second rule handles expression sort elements, which are used to implement sort aliases. In that case, the intended sort is not the expression sort itself, but the sort represented by the sort expression contained in it. We extract that sort, and create the appropriate reference. This means all references to a sort use the same model element to represent it, even if a number of different identifiers are involved. As a consequence of this, if you define the same type in multiple places, all instances will be considered equivalent, even if the literal expressions are different.

Listing 5.17: mCRL2 Bool Sort Expression

```
from textualmcrl2::Bool
to mcrl2::Bool(sort = $t)
where $t = mcrl2::BoolSort
```

Listing 5.18: mCRL2 List Sort Expression

```
from textualmcrl2::List(elementSort=$es)
to mcrl2::List(sort = $t)
where $t =
  mcrl2::ListSort(elementSort=$es.sort)
```

Listing 5.19: mCRL2 Sort Reference Expression

```
from textualmcrl2::SortRef(sortname=$n,links =
  $env)
to mcrl2::SortRef(sort=$s)
where $s in $env,
      $s = textualmcrl2::StructureSort(name =
        $n)

from textualmcrl2::SortRef(sortname=$n,links =
  $env)
with $es
to mcrl2::SortRef(sort=$s)
where $es in $env,
      $es = textualmcrl2::ExpressionSort(name
        = $n),
      $s = $es.expression.sort
```

In type rules for other mCRL2 expressions, the normalization leads to constructs as shown in Listing 5.20. The listing shows the rule for `NumberExpression` elements. In order to give these elements a type, we need to create both a Sort element and a SortExpr element to hold it. For consistency reasons, we choose to use the same element that a user would use to specify a type to represent it here.

Listing 5.20: mCRL2 Number Expression

```
from textualmcrl2::Number
to mcrl2::Number(type=$t)
where $t = mcrl2::Nat(sort=mcrl2::NatSort)
```

Listing 5.21 shows an example of a rule for more complex expressions. This particular rule defines some of the type behavior of addition, subtraction and multiplication. Because these are binary expressions, we first have to extract the types of the two subexpressions. To cover the case where the sort of a subexpression is actually a sort expression, we use the computed `sort` reference instead of the base type element. In this case, we do that by defining a variable which contains the maximum of these types. As long as that maximum is smaller than or equal to the real type, the current expression is correctly typed. In order to store the type in the model element, however, we need a SortExpr element. Because we do not want a separate rule for each possible type, we choose to use a RealSort element here, even if the actual type is different. For the purpose of the type system, this makes no difference.

Listing 5.21: mCRL2 Binary Expression Example

```
from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $t
to mcrl2::BinaryExpression(type = $et)
```

```
    where $op in set{textualmcrl2::BinaryOps::Multiplication,
                     textualmcrl2::BinaryOps::Addition,
                     textualmcrl2::BinaryOps::Subtraction},
          $t > $l.type.sort,
          $t > $r.type.sort,
          mcrl2::RealSort > $t,
          $et = mcrl2::Real(sort=$t)
```

As an example of a more complex rule, we show the rule for MapAccess expressions in Listing 5.22. An mCRL2 Map is a data structure that links keys to values, or in other words a dictionary. From an mCRL2 perspective, they are actually considered a limited form of function, in particular one that has only one parameter. MapAccess expressions are used to add new key-value pairs to a map, or to extract the key linked to a given value from a map. In the type rule, MapAccess elements are treated similarly to function calls. In particular, in the first line of the conditions, we extract the SortExpr that represents the base map expression. This will be used as the type of the complete expression if typing is successful. The next three lines extract the actual sort, which must be a HigherOrderSort. The parameter list and the result type of the function sort are stored in variables `$parameterlist` and `$valuetype` respectively. The sixth line checks if the map indeed has exactly one parameter. The seventh line extracts type of the single parameter from the list. This type is used in line 403 to check if the expression that provides the key has the correct type. The final line, line 404, checks if that type of the value expressions is compatible with the type of elements already in the map.

Listing 5.22: mCRL2 Map Expression

```
394     from textualmcrl2::MapAccess(base = $b, key = $k, value = $v)
395     with $parameterlist, $parameter, $valuetype
396     to mcrl2::MapAccess(type=$t)
397     where $t = $b.type,
398           $b.type.sort = mcrl2::HigherOrderSort
399                              (domain = $parameterlist,
400                               result = $valuetype),
401           length $parameterlist = 1,
402           $parameter in $parameterlist,
403           $k.type.sort < $parameter,
404           $v.type.sort < $valuetype
```

Evaluation

Like WebDSL, but unlike CIF, SLCO and POOSL, mCRL2 has a type checker as part of its tool set [31]. This type checker is implemented in an imperative language, and was constructed in an ad-hoc fashion. A formal definition [66] for part of the type system has been created, but the relation between the formalization and the behavior of the existing type checker is yet to be shown. The type system given in this thesis is primarily based on the formal definition, not on the implementation. Because mCRL2 is not an EMF-based language, it makes little sense to compare the performance of our type checker to the handcrafted version. If tested, we expect the existing type checker to be significantly faster. At the moment, our type checker takes several seconds to typecheck even small models, but we expect that further optimization can reduce this time. In terms of correctness, we attempted to make the types computed match the types described in the report, but because there is no official implementation of it yet, it is hard to verify if this is true in all cases.

In comparison with the formal definition, we observe that our specification is about half the size than the technical report version of the official specification. In terms of completeness, our version does not include all possible mCRL2 data expression constructs, but it does include process expressions and other constructs which are not included in the official specification. It also must be said that that the technical report includes an extensive explanation in natural language of the type system, the algorithm used to resolve applications of overloaded functions, and the decisions behind it, which naturally makes the report longer. In terms of actual rules, we observe that report uses one rule for most constructs, for a total of 21 rules for 18 constructs. In our specification, we require only one rule for most constructs, but implementation details can increase that number, resulting in 58 rules for 38 constructs. In particular, we make the rules for built-in unary and binary operators fully explicit, defining 18 rules for just those two constructs, unlike the report, which uses just one rule to cover both.

5.5 Case Study: POOSL

As mentioned in the case study introduction in Section 5, POOSL specifications are organized in layers. We will describe these layers by using examples taken from the POOSL website[5]. The *architecture layer* describes the overall structure of the system in terms of three main constructs. The first construct is the process class, which describes the basic components. The second construct is the cluster class, which groups processes and smaller clusters together. Clusters are used to introduce hierarchy into the system. The third and final construct is the channel, that describes connections between processes and clusters. All communication in POOSL is modeled as messages that pass through these chan-

[5] `http://www.ics.ele.tue.nl/\~lvbokhov/poosl/introduction/intro.html`

nels. An example[6] of an architecture layer model is shown in Listing 5.23. It describes a system consisting of two components, a sender and a receiver, operating in parallel. The sender and receiver communicate using channel c. Using the hiding operator "\", this channel is then hidden, to prevent outsiders from inserting or removing messages from the channel, thus disrupting the system. As the example shows, cluster classes can have parameters, which are commonly used to initialize components, i.e. other clusters or processes, like is done in Listing 5.23.

Listing 5.23: POOSL Cluster Example

```
cluster class SenderReceiverPair(n: Integer)
behaviour specification
    (
        s: Sender               /* instantiate
            from process class Sender */
    ||
        r: Receiver(3 * n - 4)  /* instantiate
            from process class Receiver */
    ) \{c}
```

Behavior of the processes is described in the *process layer*, in an object-oriented style. Each process is an instance of a process class that defines its methods, instance variables, ports and messages. Methods and variables adhere to common object-oriented principles, so we do not describe them here. Ports define how channels can be connected to the process, and messages define what the process can send and receive.

In Listing 5.24, we show the Receiver process class used in the architecture example as an example of a process layer model. Like cluster classes, process classes can have parameters. Unlike cluster classes, process classes can have instance variables and instance methods, defining system behavior. In the Receiver class, we have one instance variable and two instance methods. The first method, setUpTheReceptionQueue, is an initialization method, that is called every time the class is instantiated, as defined in the initial method call part of the class. The second method, receiveItems defines the actual behavior, where items are received using the ? operator, until the queue items reaches a predetermined size. Note that because POOSL process methods can have both input and output parameters, each method call has two lists of arguments. In the example, these are both empty in all cases, as indicated by the "()()" symbols, because no parameters are defined for either method.

Listing 5.24: POOSL Process Example

```
process class Receiver(itemsToReceive: Integer)
instance variables
    items: Queue                /* this variable
        will be used to store the objects received
```

[6] http://www.ics.ele.tue.nl/\~lvbokhov/poosl/language/systemspecification.html

```
        */
initial method call
    setUpTheReceptionQueue()()
instance methods
    setUpTheReceptionQueue()()
        items := new(Queue);    /* create a new queue to store objects */
        receiveItems()().

    receiveItems()()
    | anObject: Object |        /* anObject is a local variable of this method */
        while items size < itemsToReceive do
            c?m(anObject);
            items add(anObject)
        od.
```

The final layer is the *data layer*, where the data objects are described. In contrast to processes, data objects cannot send or receive messages, but are used to represent data in the system and computations on it. Like process classes, data classes can have methods, but they are restricted to data expressions, so they cannot send or receive messages, for example. An example of a data class is shown in Listing 5.25. This data class is taken from the same source[7] as the previous examples, but has no direct relation to them. The class can be used to store and use a simple finite state machine. The state machine is stored as an array, TransitionTable, linking each state to a successor. Additionally, for each state we store a binary number, as indicated by the % signs in line 9, to be used in displaying the current state of the state machine. In the class, the reset method resets the state machine to a default state. This is done by setting TransitionTable and OutputTable to a predefined value, creating a new array and using the put command to set the value of the cells one at a time. The nextState method is used to advance the state machine to the next state. The other two functions, output and displayOutput, can be used to read or display the current state of the state machine.

Listing 5.25: POOSL Data Example

```
1  data class Simple_FSM
2  extends Object
3  instance variables
4      State: Integer, TransitionTable: Array
5  instance methods
6      reset: Simple_FSM
7          /* Initialise the transition table and the output table */
```

[7] http://www.ics.ele.tue.nl/\~lvbokhov/poosl/language/dataclasses.html

```
        TransitionTable := new(Array) size(4)
            put(0, 1) put(1, 2) put(2, 3) put(3,
            0);
        OutputTable := new(Array) size(4)
            put(0,%0001) put(1, %0010)
            put(2,%0100) put(3, %1000);
        State := 0;                               /*
            Reset the internal state to 0 */
        return self.

    nextState: Simple_FSM
        State := TransitionTable get(State);  /*
            Use the transition table to compute
            the next state */
        return self.

    output: Integer
        return OutputTable get(State).         /*
            The output is found in the OutputTable
            at index State */

    displayOutput: Simple_FSM
        primitive.                                /*
            The actual implementation is provided
            in another language;*/
```

POOSL type system

In contrast with mCRL2 and WebDSL, the POOSL toolset contains no type-checker that implements the static part of the POOSL type system. This means that we can only discover the type system by looking at the documentation and at the other tools, like SHESim[8] and Rotalumis[9], which use POOSL. Additionally, the documentation gives very little information on the type system. While this is unfortunate, based on our review in Section 2.2 we expect that this situation is not uncommon among DSLs. In our SLR, we found that 76% of DSLs had no description of its type system in a paper. While this does not necessarily mean there is no type system documentation at all, we expect this to be the case for a significant fraction of those DSLs. In fact, in these cases, a formal specification can both clarify the type system, and help to expose inconsistencies and gaps in manuals and implementations. In the case of POOSL, the documentation provided online is very sparse on the matter of types. There are a number of specific comments in the documentation on the topic, but they deal mostly with some specific features of the language. For example, while

[8] http://www.ics.ele.tue.nl/\~mgeilen/shesim/index.html

[9] http://www.ics.ele.tue.nl/\~lvbokhov/poosl/rotalumis/index.shtml

mathematical operations like addition apply to to integers and reals equally, these values cannot be sensibly compared by the equality operations[10]. Unlike in many other languages, such as WebDSL and MCRL2, the integers are not part of the reals so any real is different from any integer. Thus, in the type system, it makes sense to consider any application of the equality operator to a real and an integer as a type error, because such an expression can never evaluate to anything other than false. Recall that for CIF, we had a similar restriction, but there it applied only to some, not all, expressions. In terms of our SLR, we observe that POOSL is a strongly typed language with some statically typed aspects, though in practice the models are not actually checked before they are compiled. POOSL is explicitly typed, has objects, inheritance and overloading. It does not feature type inference or type parameters. In Sections 5.5 and 5.5, we will discuss our implementation of the POOSL type system. The full formalization can be found in Appendix E.

Primary Expression

While POOSL shares many concepts with WebDSL, the metamodel is organized in a different style, and while this is not really noticeable for the user, it does impact the type system. Consider for example function calls. In WebDSL, a function call is a separate kind of expression, consisting of a function name, some arguments and possibly a base expression if the function is a member of an object. This means a function call can be combined with other expressions in arbitrary ways. In POOSL, there is instead the concept of a primary expression. A primary expression consists of a base expression, one or more method calls and a possible negation operator. If there are no method calls, the type of the primary expression is the same as the base expression. If there are method calls, the result of the base expression serves as the base object for the first method call. The result of that method call serves as the base object for the second method call, and so on. All methods are members of a class, therefore we always need a base expression to determine which instance of the class will be used to execute the method.

In our type formalism, the semantics of primary expression elements are implemented by a combination of rules, one for each situation, i.e.: only a primary expression, only method calls, or a combination of both. Because the behavior of the element is different in each case, we use a dedicated rule for each case. We show some of those rules in Listing 5.26. To be precise, we show the rules for the case where there is no negation operator. We use a separate set of rules for the case where there is a minus operator, but the only difference with the rules presented is the addition of a condition to check if the result of the expression can have the negation operator applied to it. In the version shown in Listing 5.26, the first rule handles the case where there are no method invoked. In that case, the first condition tests if there is indeed no minus operator in this element. The second and third condition check if there are no methods

[10] http://www.ics.ele.tue.nl/\~lvbokhov/poosl/resources/numbers.html

present. The final condition of the first rule states that the type of the primary expression is equal to the base expression it contains.

The next rule handles the case where the element contains method calls, but no base expression. However, this does not actually mean that any method in the model can be invoked, but in fact means that there is an implicit reference to the containing class of this primary expression. Only methods contained in the containing class can be invoked. In order to specify this, we use the `in` mechanism in EMF-TL to get the reference to the containing class, and then extract the methods from that class.

First, we get the first method that is part of this expression, and check if it is part of the base class. If this is the case, we then create a list of method pairs, using the `pairs` method, which turns a list of elements into a list of pairs of consecutive elements, as shown in line 164. In each pair, the result of the first method serves as the base object for the second method. Using a loop, we check if this connection is valid for all methods in this expression. Finally, we take the final method of the list, and extract its return type. This return type is the final type of the expression.

The third rule deals with the case where the element contains method calls and a base expression which indicates which object serves as the starting point. Overall, the rule is very similar to the previous one, except that instead of using the `in` construct, the type of the base object is extracted from the base expression.

Listing 5.26: POOSL PrimaryExpression rules

```
148 from untypedpoosl::PrimaryExpression(primary=$p,
      methodCall = $mc, minusSign = $minus)
149 with $s
150 to poosl::PrimaryExpression(type = $t)
151 where $minus = "false",
152       $s = length $mc,
153       $s = 0,
154       $t = $p.type
155
156 from untypedpoosl::PrimaryExpression(primary =
      $p, methodCall = $mc, minusSign = $minus) in
      $dc : untypedpoosl::DataClass
157 with $dcmethods, $firstmethod, $lastmethod,
      $mcpairs
158 to poosl::PrimaryExpression(type = $t)
159 where $minus = "false",
160       $p = OclUndefined,
161       $dc = untypedpoosl::DataClass(method =
            $dcmethods),
162       $firstmethod = first $mc.method,
163       $firstmethod in $dcmethods,
164       $mcpairs = pairs $mc,
      for ($mc1,$mc2) in $mcpairs:
          ($mc2.method in
            $mc1.method.returnType.method),
      $lastmethod = last $mc.method,
      $t = $lastmethod.returnType

from untypedpoosl::PrimaryExpression(primary =
  $p, methodCall = $mc, minusSign = $minus)
with $dcmethods, $firstmethod, $lastmethod,
  $mcpairs
to poosl::PrimaryExpression(type = $t)
where $minus = "false",
      $p.type = untypedpoosl::DataClass(method =
        $dcmethods),
      $firstmethod = first $mc.method,
      $firstmethod in $dcmethods,
      $mcpairs = pairs $mc,
      for ($mc1,$mc2) in $mcpairs:
          ($mc2.method in
            $mc1.method.returnType.method),
      $lastmethod = last $mc.method,
      $t = $lastmethod.returnType
```

Method Calls

In addition to the process method calls discussed in the previous section, recall that POOSL also has data methods. Calls to these data methods are typed using the rule shown in Listing 5.27. This rule is structurally similar to the rule used in CIF, as shown in Listing 5.5, WebDSL, as shown in Listing 5.11, and MCRL2 for similar constructs, but is unusual in that the reference to the method and the actual call are combined into one model element instead of two or more. Another unusual aspect is that the metamodel as created by the language designers supports a shorthand for parameter declaration, allowing multiple parameters of the same type to be declared in one element. This shorthand element has to be unfolded again to reconstruct the actual list of parameters of the function. In this rule, the method is chosen from the environment in the first three lines of the condition, lines 262 and 264. Once we have a method with the correct name, we extract the types of the parameters from it in lines 265 to 271 of the conditions, using a nested loop. We need a nested loop, because it is possible in POOSL to define multiple parameters with the same type in one shorthand declaration. Using the nested loop, we unfold this shorthand into a list of lists of types, where each type corresponds to a parameter. In line 272, we use the flatten function to convert the list of lists into just a list of types. We can then use another loop to compare the types of the arguments to the types of the parameters. If the types of all arguments fit with the types expected by the corresponding parameter, we have found a correct method for this call.

Listing 5.27: POOSL DataMethodCall rule

```
from untypedpoosl::DataMethodCall(links = $l,
  methodname = $mn, arguments = $args)
with $params, $ptypeslist, $varlists,
  $paramtypes, $argexps, $arguments
to poosl::DataMethodCall(method = $method)
where $method in $l,
      $method = untypedpoosl::NamedDataMethod
                     (parameter = $params, name =
                       $mn),
      for $param in $params,
          $ptypes in $ptypeslist,
          $vars: ($param =
            untypedpoosl::Declaration
                                  (variable = $vars),
                  for $ptype in $ptypes,
                      $var in $vars :
                      $ptype = $param.type),
      $paramtypes = flatten $ptypeslist,
      $args = untypedpoosl::ListOfExpressions
                  (expressions = $argexps),
      for $argexp in $argexps,
          $argument in $arguments,
          $paramtype in $paramtypes :
          ($argexp = untypedpoosl::Expressions
                              (expression = $argument),
           $paramtype > last $argument.type)
```

Recall that, in addition to data classes and methods, POOSL has process classes and methods. This means that in addition to data method calls, there are also process method calls. Unlike data method calls, process method calls are not part of other statements, but are directly statements in their own right. Another difference between data methods and process methods is that data methods always have exactly one returned value, while process methods can return results using output variables, but do not have to. From a type system perspective, this means that the rule for process method calls, shown in Listing 5.28, is fundamentally similar to the rule for data method calls, but we need only one rule instead of several. The rule is quite large, though, because we need to handle both input arguments and output variables, instead of only input arguments. The conditions of the rule can be divided into five parts. In the first part, consisting in the first two lines, lines 472 and 473, a method is extracted from the links feature. The next part, consisting of lines 477 to 485, collects the types of the parameters and is similar to the part of the rules for data method calls. The third part, consisting of lines 486 to 494, is similar to the second part and collects the types of the output variables. The fourth part, in lines 495 to 502, tests if the types of the arguments actually fits the types expected as parameters. An unusual complication is that the arguments of process method calls are actually syntactically lists of expressions. Each expression is evaluated in turn, but only the result of the last one is actually used for the call. This is reflected in the rule by applying the `last` function to get the relevant expression from the list. The final part, consisting of lines 503 to 511, checks if the variable provided to accept the output values created by the method.

Listing 5.28: POOSL ProcessMethodCall rule

```
466  from untypedpoosl::ProcessMethodCall(links = $l,
       methodname = $mn, inputArguments = $args,
       outputvarnames = $ovnames)
467  with $params, $ptypeslist, $paramtypes,
468       $outparams, $outptypeslist, $outparamtypes,
469       $varlists,
470       $argexps, $arguments, $ovdecs
471  to poosl::ProcessMethodCall(method = $method,
       outputVariables = $ovs)
472  where $method in $l,
473        $method = untypedpoosl::ProcessMethod(
474            inputParameter = $params,
475            name = $mn,
476            outputParameter = $outparams),
477        for $param in $params,
478            $paramvars,
479            $ptypes in $ptypeslist: (
480                $param = untypedpoosl::Declaration
481                    (variable = $paramvars),
482                for $ptype in $ptypes,
483                    $var in $paramvars :
484                        $ptype = $param.type),
485        $paramtypes = flatten $ptypeslist,
486        for $outparam in $outparams,
487            $outparamvars,
488            $outptypes in $outptypeslist: (
489                 $outparam =
                       untypedpoosl::Declaration
490                      (variable = $outparamvars),
491                 for $outptype in $outptypes,
492                    $var in $outparamvars :
493                        $outptype = $outparam.type),
494        $outparamtypes = flatten $outptypeslist,
495        $args = untypedpoosl::ListOfExpressions
496                    (expressions = $argexps),
497        for $argexp in $argexps,
498            $argument in $arguments,
499            $paramtype in $paramtypes :
```

```
              ($argexp = untypedpoosl::Expressions
                             (expression =
                               $argument),
               $paramtype > last $argument.type),
      for $ov in $ovs,
          $ovdec,
          $ovname in $ovnames,
          $outparamtype in $outparamtypes :
              ($ovdec in $l,
               $ov in $ovdec.variable,
               $outparamtype = $ovdec.type,
               $ov = untypedpoosl::Variable
                         (name = $ovname))
```

To explain this complex rule in more detail, we use a small example, consisting of Listings 5.29 and 5.30. In Listing 5.29, we show part of the declaration of a process method, where the actual implementation has been elided. This method has three input and three output parameters. Listing 5.30 shows an example call of the method, that would be typed by the rule shown in Listing 5.27. When we apply the rule, first the variables in the source element are set to their values. In this case, that means the variable `$mn` gets the value "someMethod", the variable `$args` gets as value the list of input arguments, which consists of the expression lists [qs],[qc] and [$3 * MaxQsize + 5$], and `$ovnames` gets as values the list of output variables, which are qs, $size$ and qc. Finally, the variable `$links` gets as value a list of process methods declarations from the model, including the declaration shown in Listing 5.29. Then, the conditions are applied to the list elements in the order they are placed in the list.

Listing 5.29: POOSL Process Method

```
someMethod(InA, InB: Queue, InC: Integer)(OutA,
   OutB, OutC: Object)
```

Listing 5.30: POOSL Process Method Call

```
someMethod(qs, qc, 3 * MaxQsize + 5)(qs, size, qc)
```

In the first line of the conditions, an element is selected from the list contained in `$links` variable into the `$method` variable. During computation, the engine will try all elements one by one but for the purpose of this example, we will consider only the case where the correct process method is selected here. In the next line, we compare the element in `$method` with an element pattern. Using this pattern, we can check if the element is a member of the `ProcessMethod` class, and extract the values of its features. In this case, the `$params` variable is set to the value of the `inputParameter` feature, which contains a list of input parameter declarations. In this example, there are two declarations, one for InA and InB and another for InC. The `$mn` is set to the value of the `name`, in this case the string "someMethod". Recall `$mn` was also set to that value in the initialization, so the two assignments do not conflict. In other words, this condition can only hold if the name of the process method in `$method` is the same as the name of the method name in the invocation. Finally, the `$outparams` variable is set to the value of the `outputParameter` feature, a list consisting of one declaration for *OutA*, *OutB* and *OutC*.

In the next lines, we examine the extracted input parameters, stored in `$params`, in more detail. Each declaration consists of one or more names and a type. For the purpose of method invocation, we are not interested in the names but only in the types, so we would like to have a list that contains only the types. Additionally, if a declaration contains multiple names, we would like to split it up so we have one element per parameter. In order to create that list, we define a loop over the list in `$params`, using the variable `$param` to hold each element. For example, in one instance of the loop, `$param` contains the declaration for *InA* and *InB*. For each element, we first compare it with a `Declaration` pattern, and extract the list of declared names from it, in this case, as mentioned before, *InA* and *InB*. Using a nested loop, we then create a list containing a type element for each name, resulting in $[Queue, Queue]$. As the loop is processed, the lists are then added to a new list, which is stored in `$ptypeslist`. So, after the loop is completed, `$ptypeslist` contains a list of lists of the types in the declarations, namely $[[Queue, Queue], [Integer]]$. In line 485, we use the `flatten` function to combine the lists into one list of types, which is stored in the `$paramtypes` variable, $[Queue, Queue, Integer]$.

In the next four lines, we apply the same procedure again, this time to the output parameter list. In this case, there is only one declaration, that contains three variables, *OutA*, *OutB* and *OutC*. The end result is the list $[Object, Object, Object]$, which is stored `$outparamtypes`.

Lines 497-502 describe the actual tests that verify if the types of the arguments match the types of the input parameters. To do this, we again use a loop, iterating over `$args` and `$paramtypes` at the same time. In order for this to work, those two lists must be of the same length, which they are. In the loop body, we first extract the actual expression lists from the `Expressions` elements where they are placed by the parser. We then compare a parameter type to the type of the corresponding expression list. In POOSL, the type of an expression list is the type of the last expression in the list. In the type rules, this is implemented by applying the `last` function to the list. In this example, each list has only one element, so that is trivially the last one. We then access the `type` attribute of the expression to extract its computed type. We assume that, for the purpose of this example, the variable *qs* has type *Queue*. Because the type of the first parameter is also *Queue*, this means the required widening is $Queue < Queue$, which automatically is true, because the $<$ operator in EMF-TL is reflexive.

Now that we have checked if the types of the arguments of the method call fit the input parameters, the last thing we have to do is to check if the output variables are compatible with the output parameters. Note that, at this point, we have not yet actually identified which variables will be used for the output parameters, we only have access to their names. Both the computation of the referenced variables and the checking of the types is implemented in one loop, that iterates over the output variables from `$ovs` and the output parameter types from `$outparamtypes`. For each variable reference, we extract a variable from `$links`, and check if the name matches the references. Once we have found the variable with the correct name, we extract its type from the declaration. Similarly to the previous loop, we then compare the type of the variable to the type of the output parameter. In the example, if we assume variables *OutA*, *OutB* and *OutC* are all declared with type *Object*, these comparisons can all be resolved without problems.

Once all conditions have been satisfied, the relevant result are linked to the corresponding features of the target element, as defined in the target element of the rule. By doing this, we allow rules being applied to other elements to access that information. In this case, the information computed is the method invoked and the output variables that will be used to store the result. Because POOSL process methods do not have a single return type, we do not not actually compute a type for this element.

Evaluation

Unlike WebDSL and mCRL2, but like CIF and SLCO, POOSL does not have an existing type checker. In contrast to CIF, but like SLCO, the language was also already completely designed before we made our specification. In terms of complexity, POOSL lies between SLCO and CIF. It has more types than SLCO, and complex object types in particular, but it does not have as many types, and particularly compound types, as CIF. The complexity of the POOSL type system lies mainly in the presence of complex expression elements. The elements combine several concepts that are represented by separate constructs in other languages into one. In EMF-TL, this results in rules larger than we see for other languages, but because they themselves combine a number concepts that would be split over several elements in many other languages, we feel they are still understandable and maintainable. In terms of completeness, our specification is based on the way types are described in the documentation, extended with what we observe from the examples given in the documentation referred to earlier. Because the existing POOSL tools that would normally handle typed models, like the Rotalumis execution engine, are not actually equipped to use this information, we were not able to test how well the types computed match the expectations of the tools.

In terms of performance, we cannot compare our implementation to any existing POOSL type checker, because no other exists. We can observe however, that our generated POOSL type checker is equal in speed to other type checkers generated based on our specifications, checking models with tens of elements in a few seconds, which means the performance is useable for the scale of model we tested.

5.6 Conclusion

In this chapter we looked at four domain specific languages from the perspective of EMF-TL. We specified the type systems of the languages either whole or partially. Here, we will first recap the specific type system that stood out for each language.

CIF

In the first case study, we described a type system specification of CIF in EMF-TL, as a continuation of our case study for MSOS in Chapter 3.9. We considered examples from all parts of an EMF-TL specification, and explain how they were applied to CIF. In particular, we looked at the restrictions on numeric type widening we designed in Section 3.5, and found that we did not need the extra types introduced there to implement the same behavior. Recall that we essentially used these types to indicate that widening was possible. The disadvantage of this approach was that these types were considered undesirable outside the typing phase, creating a need for additional processing to make the model suitable for further processing. By using constraints in the definition of the rules of numeric constants in combination with a strategy to express our preferred type, we can create the same flexibility without introducing new types, thus ensuring the created model is useable in further processing directly.

We have also shown how function calls can be implemented in EMF-TL in Listing 5.5. Because function types can have arbitrary numbers of parameters, this is a prime situation where loop constructs are very useful. We show how the EMF-TL, we can extract the types of the parameters and the return type of the function from any function type. We can then compare the types of the provided arguments to the expected types, to determine if the function call is valid. If there are multiple options, we can use the strategy functionality to select the preferred function from the available alternatives.

WebDSL

In the WebDSL case, we specifically looked at what impact quantified references and inheritance have on our way of handling scoping. For the first concept, quantified references, we essentially choose a filtering approach. We observed that every quantified reference can be split into two parts: an 'outer' reference that defines an environment, and another, 'inner', part that references something in that environment. By reusing the rules for basic references, we get an overestimate of the possible targets for the reference. The outer reference can then filter out the targets that are not part of the correct environment. This results in a very clean and simple specification, using only one rule of eight lines, that makes full use of the strengths of the constraint solver.

For the second concept, overloading, we heavily used constraint solver concepts, in particular the functionality where a set of values can be assigned to a variable, that is refined at a later stage. The core idea of overloading is based on the fact that some functionality can be implemented in multiple ways, each with their own advantages and disadvantages. By gathering all implementations under one identifier, thus overloading it, we can shift the burden of selecting the best implementation from the programmer to the type checker. The main problem created this way for the type checker lies in the need to arrive at a single solution, while keeping all options open as long as possible. By using constraints to describe the rules of the type system, we can achieve this. The constraint solver will keep track of all possible values for as long as they remain valid. After all possible type rules have been applied, we arrive at a set of possible typings for the whole model. Using techniques from optimization, we then select the best possible result from this set.

mCRL2

As we worked on the mCRL2 case study, we discovered that the most unusual part of this type system is the sort normalization required. While other type systems use references to types, these references are usually limited to specific kinds of types, and each type has only one name. In mCRL2, multiple names can be defined for any type of sort. We addressed this feature by creating a clear separation between sort expressions and sorts. For each sort expression, we defined type rules that describe how the underlying sort can be computed. These computed values are then used as basis for the rest of the type system, ensuring that all ways of referencing the same sort are indeed treated as equals.

The mCRL2 type system also features element transformations. Because invocations of atomic actions and processes use the same syntax, it is not possible for the parser to separate them into different element classes. Instead, during typing, we discover the details of the target of the reference, and use that to create either and action invocation or a process invocation in the result model.

POOSL

In the final case study, we looked at the type system of POOSL and how its static part can be specified in EMF-TL. We choose POOSL as an example of an object-oriented language that is structured differently than WebDSL, allowing us to compare our the structure of two languages and its effect on our specification, as is primarily done in Section 5.5. We found that the main difference between the two languages lay in the way the way expressions were structured. in WebDSL, all expressions are considered equal, and they can be nested in arbitrary combinations. In POOSL, binary and unary expressions are handled separately, and function calls can only be placed in specific places. This results in rules that are similar in structure, but different in implementation.

Overall results

Overall, though the implementation was more complex and actually required a change in our metamodel of mCRL2, the type system could be implemented without modifying or extending EMF-TL. While we focussed on a subset of mCRL2 that could be handled by our parser, we believe the remaining part could be implemented without any problems if desired, and if a more advanced parser was used to implement the language syntax. This demonstrates EMF-TL as a viable answer to RQ 3: “What is the specification formalism most suited for describing the type systems of DSLs?".

Chapter 6

Conclusions

6.1 Contributions

In recent years, the concept of DSLs has become more popular. DSLs are a fundamental part of MDE, because they are used to create the models that define systems and serve as input for model transformations. In order to construct a DSL, we need, as for any formal language, to specify its syntax and semantics. While much work has been done on this kind of specification, one area has remained underdeveloped, namely specification of static semantics, in particular type systems. The central research question of this thesis is therefore:

RQ. *How can DSL type systems be specified in an understandable, formal and evolvable way?*

To answer this research question, we need to find one or more formalisms that can be used to define type systems, and evaluate their usefulness for DSLs. We decided we first needed to know more about DSLs and their type systems. We conducted a SLR aimed at discovering what type system properties and features are common among DSLs. We found that a significant number of DSLs were typed, and most DSL type systems were static and strong. Based on these findings, we selected MSOS as our first semantics formalism. MSOS is based on the well-known SOS formalism, a rule-based operational semantics formalism, which has been used for the static semantics of GPLs. We choose a rule-based formalism because it naturally supports the judgments needed for strong type systems, and an operational formalism because it supports type checking as a static phase separately from dynamic semantics. MSOS differs from SOS in the presence of modularity features, which makes specifications more reuseable and evolvable. Because DSLs are limited by nature, we consider the ability to easily construct new DSLs or adapt existing DSLs to new circumstances an important use case of a type system specification. We defined the type system of the DSL Chi in MSOS, and created a PyKE-based implementation by using a type checker generator. While we achieved some initial success with this approach, we eventually decided MSOS was not as suitable for type system specification as we initially hoped. In particular, we found specifications were closely tied with specific input and output formats, limiting interoperability with other components in DSL toolsets. We also found type system constructs creating or dealing with ambiguity were hard to express in an understandable way. Thus, we decided to create a new formalism ourselves, called EMF-TL. In contrast with MSOS, EMF-TL makes use of the Ecore format provided by the EMF framework as a generic way to interact with other tools, and provides more flexible constructs that make type system specification easier, clearer and more convenient, as shownin Section 4.3. We formally defined the semantics of EMF-TL, and created a generator that creates type checkers based on specifications. We then conducted a number of case studies where we defined (part of) the type systems of several languages, namely CIF, WebDSL, mCRL2 and POOSL, in EMF-TL. We choose these languages as example DSLs to cover the most common and important DSL type system properties we discovered in our SLR 2.2. For each language, we discussed several type system features of that language and how they were implemented in EMF-TL. With these case studies, we demonstrated that EMF-TL provides an understandable, formal and evolveable method for defining type systems, thus providing an answer for our research question. In the remainder of this section, we will discuss our more detailed research subquestions, and how we addressed them.

DSL type system features

While DSLs have increased in popularity recently, the concept has been around for some time and a significant number of DSLs have already been developed over the years. Each one of those DSLs has static semantics to some extent, whether formally defined or not. When we want to consider the suitability of a given formalism for DSL type system specification, these DSLs can provide valuable data on what kind of properties we should look at. This led to the following research question:

RQ 1. *What are common features of DSL type systems?*

To answer this question, we first had to collect data on DSLs and their properties. Because many DSLs are designed and used by specific companies, groups or even individuals and not intended for public use, we cannot expect to get a complete picture of all existing DSLs. Instead, we choose to do a SLR to collect a representative sample. For this review, we looked at information on DSLs published in scientific papers. We found a total of 497 DSLs in our initial search. We looked at a number of properties, and described our findings in Section 2.2. Our conclusions were that a significant number, 279 of the 497, of the DSLs are typed. Based on the information available, we then selected 173 of the 279 typed DSLS as subjects of a more detailed analysis, based a number of properties we expected DSL type systems to have. We found that a number of basic properties are indeed shared by most DSLs. For example, many DSLs, 139 of 173 considered, are statically and strongly typed. More advanced features like object-orientation and type inference were present, but were rarer. Based on this, we concluded that our type system specification formalism should focus on statically, strongly typed languages. If possible, it should support features like overloading and type inference as well, but those features are less common, and hence, less important.

Another factor in the suitability of type system specification formalisms lies in the people who interact with it. For GPLs, the vast majority of people who interact with the type system will be users of the language. For DSLs, the number of users is typically much lower, making the other stakeholders relatively more important. Overall, these considerations led to the following research question:

RQ 2. *How can DSL type systems specification help stakeholders reach their goals?*

We discussed our vision of the priorities of the different stakeholders in Section 2.3. The stakeholders are first and foremost the users of the language, but also the language designers and the language implementers. Based on our discussions of the relation of the stakeholders with type system specifications, we formulated several requirements that the specification formalism should meet.

Based on the requirements, we have to select one or more suitable formalisms to investigate further. There are a number of choices to make, because our requirements do not specify if the language should use primarily textual or graphical syntax. This led to the following, basic, formalization of our next research question:

RQ 3. *What is the specification formalism most suited for DSL type systems?*

The first formalism we selected was MSOS. MSOS is an operational semantics formalism based on the well-known SOS language. We explored the possibilities of applying MSOS to the type system of the CIF language in Chapter 3. While we initially found MSOS flexible and intuitive, later on we found these features were lost as we tried to incorporate more complex type system concepts. Based on those experiences, we decided to develop a new specification formalism, EMF-TL. EMF-TL is introduced in Chapter 4, and further described in several case studies, described in Chapter 5. Using these studies, we tried to demonstrate that EMF-TL is a valid answer to this research question.

When we were working with MSOS, we decided to focus on one particular requirement where we expect a declarative formalism like MSOS to have an advantage: evolvability. We consider DSLs to be more likely to evolve than GPLs, both because DSLs are limited in their expressiveness, increasing the chance a new requirement can only be implemented by extending the language, and because the number of users is lower, decreasing the cost of changing the language. By using a more abstract, declarative specification over a more direct implementation, we expect we can both evolve the type system and adapt to evolution of other components more easily. Overall, this leads to the following research question:

RQ 4. *How can DSL type systems specification assist the process of language evolution?*

In order to answer this question, we observe that language evolution can take many forms and take take place in all components of a language. Because the type checker essentially serves as a bridge between other components, there are more opportunities for it to be affected by evolution. For example, if a new shorthand notation is introduced in the parser for some elements, the type checker has to be updated to handle these new cases. A particular issue here is that if the implementation of the type checker is updated separate from the specification, there is a significant risk the implemented semantics will deviate from the specified semantics.

To prevent this from occurring, we decided to study the concept of type checker generation, to minimize this risk. We first applied this concept to MSOS, as described in Chapter 3.9. We found the type checker generator made experimentation much easier, but the resulting specification was significantly influenced by the details of the in- and output interfaces with parsers and execution environment respectively. Combined with complicated MSOS rules required to implement the desired features, we found the benefits from generation less than they could be, because specifications could not be easily reused for different configurations. Based on these experiences, we decided to develop a new generator for our new formalism, EMF-TL. The details of this generator are described in Chapter 4. By constructing our formalism around the EMF framework, we reduced the dependency on specific parsers and other components, and by incorporating a constraint solver component in the generated type checker, we made it possible to reduce the complexity of the specifications. Overall, we conclude that generation significantly reduces the effort to keep a type checker consistent with its specification. Any evolution in the specification will be reflected in the generated type checker directly and consistently.

In addition to implementation aspects of type system specification, they can also be used as documentation of type systems. However, a specification is only useful in this regard if it is clear and understandable. This means our formalism needs the right constructs to allow DSL type system concepts to be expressed effectively and concisely. This leads to our last research question:

RQ 5. *How can DSL type system features best be expressed in our chosen formalism?*

Fundamentally, this relates closely to our survey of DSL type systems from Section 2.2, as we cannot expect a formalism to express all imaginable semantics with equal proficiency, nor can we cover them all in this thesis. Thus, we decided to look at some of the most popular type system features, as discovered in our SLR. For each of these features, we looked for a case study that contains that feature and focussed on it during our type system discussion of that case. For example, we choose WebDSL as an example object-oriented language, and discussed how we implemented this in our type system formalism in Section 5.3. Overall, we found type rules most effective when focussed on conditions applying to one element at a time. By combining several of these rules, we can create flexible type systems that work intuitively. If we need to refer to multiple elements in one rule, the rules tends to become much more complex, and more rules are often needed to cover all possible cases.

6.2 Directions for future research

In this thesis, we present research into, and a formalism for, type systems for DSLs. This section addresses possible directions for future research in this area.

The survey we did on domain specific languages and their type systems is extensive, but questions remain on how representative our DSLs are compared to the complete population of existing DSLs. Due to limitations in the time and resources available, we had to restrict our survey to information on DSLs that can be found in peer-reviewed papers. While we can expect DSLs created in academic context to be published in this way, this is definitely not the case for industrial DSLs. This means that if there is a difference between academic and industrial DSLs, it will not show up in our survey. Further surveys that collect material from other sources would be very beneficial both as a basis of comparison, in addition to increasing the number of covered DSLs. Additionally, our survey focussed on the theoretical aspects of DSL type systems. In particular, we did not consider the implementations of the DSL type systems we found, looking only at the information given in papers, because the sheer number of DSLs involved made doing more in-depth investigations impractical. A more focused study could look at actual implementations of a more limited set of DSLs and compare them to their descriptions, to discover how well descriptions match implementations.

In Chapters 3 and 3.9, we considered MSOS as a type system specification language. While we eventually concluded MSOS was not as suitable as a type system specification formalism as we hoped, it is still possible to use MSOS as such. To recap, one issue we had with MSOS is that it was unclear how to effectively establish a connection between the type system specification and the structure of its input and output. This makes it much harder to check the specification for inconsistencies, because we do not know what the type checker can expect as input or what constitutes a valid output. Another issue we had with MSOS lay in the lack of the ability to disambiguate cases where there were multiple valid solutions. In EMF-TL, we solved this through the introduction of special widening and strategy operators, allowing designers to both specify possible choices and to select one choice to be the final answer.

In response to our experience with MSOS, in Chapter 4 we introduced a new formalism, EMF-TL, as an alternative that does not suffer from the same drawbacks as MSOS. We defined formal semantics for EMF-TL, and created a prototype generator that creates type checkers. While the generated type checkers are useable, performance was not a major consideration in the implementation of the prototype generator and further optimization should be certainly possible. Another direction for future work in type checker generation could lie in the connection of EMF-TL with the EMF framework. While this connection offers considerable benefits in allowing the generated type checkers to interface with a variety of tools, not every tool works with EMF, so it might be worth considering extending the language and/or the generator to deal with other modeling frameworks.

In addition to improving the generator, the language itself can be improved further. We did case studies to test and demonstrate the effectiveness of the language in dealing with common type system features, but further case studies might well show that the language is lacking some constructs required to fully specify the desired type system. For example, EMF-TL is not intended to be Turing-complete, so any language for which the type system is Turing-complete, can probably not be expressed fully in EMF-TL. At this point, we do not consider that a major flaw in the language, because DSLs with Turing-complete type systems are very rare, but other limitations might be evaluated differently. In a different direction, one case study that would be interesting is to describe the type system of EMF-TL itself. We anticipate no particular difficulties in this, because the EMF-TL type system is quite basic, but it would be valuable to define this part of the language more formally, and allow us to simplify the language by removing some disambiguation constructs.

Furthermore, another direction we could consider other forms of static analysis that EMF-TL is suitable for. While primarily designed with the idea of computing types, EMF-TL can in theory compute all kinds of values. As long as an analysis can be defined using a structure similar to a type system, it can be defined in EMF-TL. In particular, we could consider data and control flow. For example, data flow analysis could be used to discover if some computed values are never actually used, which strongly suggests computing them can be avoided, thus increasing performance. The results of these are primarily used for optimization and code generation rather than error detection, but those are relevant concerns even for DSLs.

We have to be careful, however, not to overextend the language. Our language focusses on computing local properties, based on information contained in each model element itself or in elements directly related to it. In contrast, some kinds of analysis, like name binding, strongly relate to the global structure of the models involved. As discussed in Section 4.3, we currently use a handcrafted model transformation to link references to potential targets, converting a global problem to a more local one. While suitable for the scope of this thesis, this is not the most user-friendly and maintainable solution. It might be better to use a DSL, like for example NBL [70], that covers this area instead.

This leads to a more general point. While EMF-TL is designed to be a DSL for type systems, that does not mean it can cover all type systems imaginable. There might be some type systems that are better served by designing a completely new language, to deal with their particular structure. A prime example are DSLs with dynamic type systems: EMF-TL does not have the constructs to describe the interaction with the dynamic semantics that would be required. Adding the needed constructs would require major changes to not just the language, but also its implementation. This suggest that extending EMF-TL as opposed to developing a separate type system language would not offer many advantages, while likely reducing EMF-TL's ease of use for static type systems.

Overall, we expect a number of languages and tools will be needed to cover all use cases. We can draw experience on this from some other components of DSL tool sets, like those for model creation and editing tools. For textual languages, a large number of parser generators have been created over the years, and similarly a number of tools have been developed for graphical DSLs. Until now, we have not seen many similar tools for further processing of models, like type checking and other forms of analysis.

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Prentice Hall, 2006.

[2] Aiken, A., and Wimmers, E. I. Type inclusion constraints and type inference. In *FPCA* (New York, NY, USA, 1993), ACM, pp. 31–41.

[3] Alam, M., Hafner, M., Breu, R., and Unterthiner, S. A framework for modelling restricted delegation of rights in the SECTET. *Comput. Syst. Sci. Eng. 22*, 5 (2007), 142–151.

[4] van Amstel, M. F., van den Brand, M. G. J., and Engelen, L. J. P. An exercise in iterative domain-specific language design. In *IWPSE-EVOL* (New York, NY, USA, 2010), ACM, pp. 48–57.

[5] Andova, S., van den Brand, M. G. J., Engelen, L. J. P., and Verhoeff, T. MDE Basics with a DSL Focus. In *SFM* (2012), M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds., vol. 7320 of *Lecture Notes in Computer Science*, Springer, pp. 21–57.

[6] Apt, K. R., and Wallace, M. *Constraint logic programming using ECL*^{i}*PS*^{e}. Cambridge University Press, 2007.

[7] Attali, I. Compiling TYPOL with Attribute Grammars. In *PLILP* (1988), P. Deransart, B. Lorho, and J. Małuszyński, Eds., vol. 348 of *Lecture Notes in Computer Science*, Springer, pp. 252–272.

[8] Balland, E., Brauner, P., Kopetz, R., Moreau, P., and Reilles, A. Tom: Piggybacking Rewriting on Java. In *RTA* (2007), F. Baader, Ed., vol. 4533 of *LNCS*, Springer, pp. 36–47.

[9] van Beek, D. A., Collins, P., Nadales, D. E., Rooda, J. E., and Schiffelers, R. R. H. New Concepts in the Abstract Format of the Compositional Interchange Format. In *ADHS* (Zaragoza, Spain, 2009), A. Giua, C. Mahuela, M. Silva, and J. Zaytoon, Eds., pp. 250–255.

[10] van Beek, D. A., Hofkamp, A. T., Reniers, M. A., Rooda, J. E., and Schiffelers, R. R. H. Syntax and consistent equation semantics of Chi 2.0. SE Report 2008-01, Eindhoven University of Technology, 2008.

[11] van Beek, D. A., Man, K. L., Reniers, M. A., Rooda, J. E., and Schiffelers, R. R. H. Syntax and Consistent Equation Semantics of Hybrid Chi. *JLAP 68*, 1-2 (2006), 129–210.

[12] van Beek, D. A., Reniers, M. A., Schiffelers, R. R. H., and Rooda, J. E. Foundations of a Compositional Interchange Format for Hybrid Systems. In *HSCC* (2007), A. Bemporad, A. Bicchi, and G. C. Buttazzo, Eds., vol. 4416 of *Lecture Notes in Computer Science*, Springer, pp. 587–600.

[13] Bergstra, J. A., and Klop, J. W. Process Algebra for Synchronous Communication. *Information and Control 60*, 1-3 (1984), 109–137.

[14] Bettini, L. A DSL for writing type systems for Xtext languages. In *PPPJ* (2011), ACM, pp. 31–40.

[15] Bézivin, J. Model driven engineering: an emerging technical space. In *GTTSE* (2006), Springer-Verlag, pp. 36–64.

[16] Boyer, R. S., and Moore, J. S. A mechanical proof of the Turing completeness of pure LISP. In *Automated Theorem Proving* (1984), American Mathematical Society, pp. 133–167.

[17] Braga, C. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics.* PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, 2001.

[18] Braga, C., Haeusler, E. H., Meseguer, J., and Mosses, P. D. Mapping Modular SOS to Rewriting Logic. In *LBPST* (2002), M. Leuschel, Ed., vol. 2664 of *LNCS*, Springer, pp. 262–277.

[19] van den Brand, M. G. J., Heering, J., Klint, P., and Olivier, P. A. Compiling Rewrite Systems: The ASF+SDF Compiler. *ACM TOPLAS 24*, 4 (2002), 334–368.

[20] van den Brand, M. G. J., Iversen, J., and Mosses, P. D. An Action Environment. *SCP 61*, 3 (2006), 245–264.

[21] van den Brand, M. G. J., van der Meer, A. P., and Serebrenik, A. Type Checking Evolving Languages with MSOS. In *Semantics and Algebraic Specification* (2009), J. Palsberg, Ed., vol. 5700 of *LNCS*, Springer, pp. 207–226.

[22] van den Brand, M. G. J., van der Meer, A. P., Serebrenik, A., and Hofkamp, A. T. Formally specified type checkers for domain specific languages: experience report. In *LDTA* (New York, NY, USA, 2010), ACM, pp. 12:1–12:7.

[23] Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. Stratego/XT 0.17. A language and toolset for program transformation. *SCP 72*, 1–2 (2008), 52–70.

[24] Bruce, K. B., Crabtree, J., Murtagh, T. P., van Gent, R., Dimock, A., and Muller, R. Safe and decidable type checking in an object-oriented language. *SIGPLAN Not. 28*, 10 (1993), 29–46.

[25] Burella, J., Rossi, G., Luna, E. R., and Grigera, J. Dealing with Navigation and Interaction Requirements Changes in a TDD-Based Web Engineering Approach. In *XP* (2010), A. Sillitti, A. Martin, X. Wang, and E. Whitworth, Eds., vol. 48 of *Lecture Notes in Business Information Processing*, Springer, pp. 220–225.

[26] Bürger, C., Karol, S., and Wende, C. Applying attribute grammars for metamodel semantics. In *FML* (New York, NY, USA, 2010), ACM, pp. 1:1–1:5.

[27] Chalub, F. An implementation of modular SOS in Maude. Master's thesis, Universidade Federal Fluminense, Brazil, 2005.

[28] Christensen, N. H. *Domain-specific languages in software development – and the relation to partial evaluation*. PhD thesis, DIKU, Dept. of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, 2003.

[29] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. The Maude 2.0 System. In *RTA* (2003), R. Nieuwenhuis, Ed., vol. 2706 of *LNCS*, Springer-Verlag, pp. 76–87.

[30] Cleve, A., Henrard, J., and Hainaut, J.-L. Co-transformations in Information System Reengineering. *ENTCS 137*, 3 (2005), 5–15.

[31] Cranen, S., Groote, J. F., Keiren, J. J. A., Stappers, F. P. M., de Vink, E. P., Wesselink, W., and Willemse, T. A. C. An Overview of the mCRL2 Toolset and Its Recent Advances. In *TACAS* (2013), N. Piterman and S. A. Smolka, Eds., vol. 7795 of *Lecture Notes in Computer Science*, Springer, pp. 199–213.

[32] Deransart, P., Ed-Dbali, A., and Cervoni, L. *Prolog: the standard: reference manual*. Springer, London, UK, 1996.

[33] van Deursen, A., and Klint, P. Little languages: little maintenance? *Journal of Software Maintenance 10*, 2 (1998), 75–92.

[34] van Deursen, A., Klint, P., and Visser, J. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Not. 35*, 6 (2000), 26–36.

[35] Devanbu, P. T., Rosenblum, D. S., and Wolf, A. L. Generating Testing and Analysis Tools with Aria. *ACM TOSEM 5*, 1 (1996), 42–62.

[36] Dijkstra, A., and Swierstra, S. D. Typing Haskell with an Attribute Grammar. In *AFP* (2004), V. Vene and T. Uustalu, Eds., vol. 3622 of *Lecture Notes in Computer Science*, Springer, pp. 1–72.

[37] Doh, K. G., and Mosses, P. D. Composing programming languages by combining action-semantics modules. *SCP 47*, 1 (2003), 3–36.

[38] Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., and Hanus, M. Xbase: implementing domain-specific languages for Java. In *GPCE* (2012), K. Ostermann and W. Binder, Eds., ACM, pp. 112–121.

[39] Efftinge, S., and Völter, M. oAW xText: A framework for textual DSLs. In *Eclipsecon Summit Europe 2006* (2006).

[40] Ekman, T., and Hedin, G. Modular Name Analysis for Java Using JastAdd. In *GTTSE* (2006), R. Lämmel, J. Saraiva, and J. Visser, Eds., vol. 4143 of *LNCS*, Springer, pp. 422–436.

[41] Favre, J. M. Meta-Model and Model Co-evolution within the 3D Software Space. In *ELISA* (2003).

[42] Felleisen, M. On the expressive power of programming languages. *SCP 17*, 13 (1991), 35–75.

[43] Flanagan, C. Hybrid type checking. In *POPL* (2006), J. G. Morrisett and S. L. P. Jones, Eds., ACM, pp. 245–256.

[44] Fowler, M. *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[45] Frederiksen, B. Pyke, 2009. http://pyke.sourceforge.net.

[46] Friedman-Hill, E. *Jess in Action: Java Rule-Based Systems*. Manning Publications, 2002.

[47] Garwick, J. V. Programming Languages: GPL, a truly general purpose language. *Commun. ACM 11*, 9 (1968), 634–638.

[48] Ghosh, D. *DSLs in Action*, 1st ed. Manning Publications Co., Greenwich, CT, USA, 2010.

[49] Gosling, J., Joy, B., Steele, G. L., Bracha, G., and Buckley, A. *The Java Language Specification, Java SE 7 Edition*. Java Series. Pearson Education, 2013.

[50] Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M., and Tolvanen, J.-P. DSLs: the good, the bad, and the ugly. In Harris [54], pp. 791–794.

[51] Grimm, R., Harris, L., and Le, A. Typical: taking the tedium out of typing. Tech. Rep. TR2007-904, New York University, 2007.

[52] Groote, J. F., Mathijssen, A., Reniers, M. A., Usenko, Y. S., and van Weerdenburg, M. The Formal Specification Language mCRL2. In *MMOSS* (2006), E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, Eds., vol. 06351 of *Dagstuhl Seminar Proceedings*, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[53] Groote, J. F., and Ponse, A. The Syntax and Semantics of μCRL. In *ACP*, A. Ponse, C. Verhoef, and S. F. M. Vlijmen, Eds., Workshops in Computing. Springer London, 1995, pp. 26–62.

[54] Harris, G. E., Ed. *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA* (2008), ACM.

[55] Harrison, M. A. *Introduction to Formal Language Theory*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[56] Hedin, G., and Magnusson, E. JastAdd-Ůa Java-based system for implementing front ends. *ENTCS 44*, 2 (2001), 59 – 78.

[57] Hofkamp, A. T., and Rooda, J. E. Chi 1.0 reference manual. SE Report 2008-04, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2008.

[58] Hofkamp, A. T., and Rooda, J. E. Chi 2.0 language reference manual. SE Report 2008-02, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, 2008.

[59] Hudak, P. Building Domain-Specific Embedded Languages. *ACM Comput. Surv. 28*, 4es (1996), 196.

[60] ISO JTC 1/SC 32. *ISO/IEC 9075:1992 Information technology - Database languages - SQL.* ISO,Geneva,Switzerland, 1992.

[61] ISO JTC 1/SC 32. *ISO/IEC 9075-1:2008 Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework).* ISO,Geneva,Switzerland, 2008.

[62] Jaffar, J., and Maher, M. J. Constraint logic programming: a survey. *The Journal of Logic Programming 19Ũ20, Supplement 1* (1994), 503–581. Special Issue: Ten Years of Logic Programming.

[63] JetBrains.com. Meta Programming System, 2012. http://www.jetbrains.com/mps/index.html.

[64] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. ATL: A model transformation tool. *SCP 72*, 1-2 (2008), 31–39.

[65] Jürgens, E., and Pizka, M. Tool-Supported Multi-Level Language Evolution. In *SSVM* (Helsinki, Finland, 2007), T. Männistö, E. Niemelä, and M. Raatikainen, Eds., no. 3 in Helsinki University of Technology Software Business and Engineering Institute Research Reports, pp. 48–67.

[66] Keiren, J. J. A., and Reniers, M. A. Type checking mCRL2. Computer Science Report 11-11, Eindhoven University of Technology, 2011.

[67] Kitchenham, B. Procedures for Performing Systematic Reviews. Tech. Rep. TR/SE-0401, Keele University, 2004.

[68] Klint, P., van der Storm, T., and Vinju, J. On the impact of DSL tools on the maintainability of language implementations. In *LDTA* (New York, NY, USA, 2010), ACM, pp. 10:1–10:9.

[69] Knaus, C. Y. Essential programming paradigm. In Harris [54], pp. 823–826.

[70] Konat, G. D. P., Kats, L. C. L., Wachsmuth, G., and Visser, E. Declarative Name Binding and Scope Rules. In *SLE* (2012), K. Czarnecki and G. Hedin, Eds., vol. 7745 of *Lecture Notes in Computer Science*, Springer, pp. 311–331.

[71] Kosar, T., López, P. E. M., Barrientos, P. A., and Mernik, M. A preliminary study on various implementation approaches of domain-specific language. *Information & Software Technology 50*, 5 (2008), 390–405.

[72] Kosar, T., Mernik, M., and Carver, J. C. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering 17*, 3 (2012), 276–304.

[73] Kosar, T., Oliveira, N., Mernik, M., Pereira, M. J. V., Crepinsek, M., da Cruz, D. C., and Henriques, P. R. Comparing general-purpose and domain-specific languages: An empirical study. *Comput. Sci. Inf. Syst. 7*, 2 (2010), 247–264.

[74] Lämmel, R., and Wachsmuth, G. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In *LDTA* (2001), M. G. J. van den Brand and D. Parigot, Eds., vol. 44 of *Electronical Notes in Theoretical Computer Science*, Elsevier Science.

[75] Lerner, B. S., Flower, M., Grossman, D., and Chambers, C. Searching for type-error messages. *SIGPLAN Not. 42* (2007), 425–434.

[76] Levin, M. Y., and Pierce, B. C. TinkerType: a language for playing with formal systems. *JFP 13*, 2 (2003), 295–316.

[77] Liskov, B., and Zilles, S. Programming with abstract data types. *SIGPLAN Not. 9*, 4 (1974), 50–59.

[78] van der Meer, A. EMF-TL implementation code. `http://dx.doi.org/10.5281/zenodo.10870`, 2014.

[79] van der Meer, A. Results of DSL SLR. `http://dx.doi.org/10.4121/uuid:6ed87dcc-bce3-4ccd-9652-e8d7145edaef`, 2014.

[80] Merkle, B. Textual modeling tools: overview and comparison of language workbenches. In *SPLASH* (New York, NY, USA, 2010), ACM, pp. 139–148.

[81] Mernik, M. *Formal and Practical Aspects of Domain-specific Languages: Recent Developments*. Information Science Reference, 2013.

[82] Mernik, M., Heering, J., and Sloane, A. M. When and how to develop domain-specific languages. *ACM Comput. Surv. 37*, 4 (2005), 316–344.

[83] Mosses, P. D. *Action Semantics*, vol. 26 of *CTTCS*. Cambridge University Press, 1992.

[84] Mosses, P. D. Foundations of Modular SOS. In *MFCS* (1999), M. Kutylowski, L. Pacholski, and T. Wierzbicki, Eds., vol. 1672 of *LNCS*, Springer, pp. 70–80.

[85] Mosses, P. D. Modular structural operational semantics. *JLAP 60-61* (2004), 195 – 228.

[86] Pagel, M., and Brörkens, M. Definition and Generation of Data Exchange Formats in AUTOSAR. In *ECMDA-FA* (2006), A. Rensink and J. Warmer, Eds., vol. 4066 of *Lecture Notes in Computer Science*, Springer, pp. 52–65.

[87] Pettersson, M. *Compiling Natural Semantics*, vol. 1549 of *LNCS*. Springer, 1999.

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

[89] Plotkin, G. A structural approach to operational semantics. *JLAP 60–61* (2004), 17–139.

[90] Plotkin, G. D. A Structural Approach to Operational Semantics. Tech. Rep. DAIMI FN-19, University of Aarhus, 1981.

[91] Plump, D. The Graph Programming Language GP. In *CAI* (2009), S. Bozapalidis and G. Rahonis, Eds., vol. 5725 of *LNCS*, Springer, pp. 99–122.

[92] Pottier, F. A framework for type inference with subtyping. *SIGPLAN Not. 34* (1998), 228–238.

[93] van der Putten, P. H. A., and Voeten, J. P. M. *Specification of reactive hardware/software systems*. PhD thesis, Eindhoven University of Technology, 1997.

[94] Rahmani, T., Oberle, D., and Dahms, M. An Adjustable Transformation from OWL to Ecore. In *MoDELS* (2010), D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds., vol. 6395 of *Lecture Notes in Computer Science*, Springer, pp. 243–257.

[95] Rusu, V. Embedding domain-specific modelling languages in Maude specifications. *SSM* (2012), 1–23.

[96] Schmidt, D. A. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[97] Sipser, M. *Decidability*. PWS Publishing, 1997, pp. 151–170.

[98] Solar-Lezama, A., Rabbah, R. M., Bodík, R., and Ebcioglu, K. Programming by sketching for bit-streaming programs. In *PLDI* (2005), V. Sarkar and M. W. Hall, Eds., ACM, pp. 281–294.

[99] Spinellis, D. Notable design patterns for domain-specific languages. *Journal of Systems and Software 56*, 1 (2001), 91–99.

[100] Sprinkle, J., Mernik, M., Tolvanen, J.-P., and Spinellis, D. Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer? *IEEE Software 26*, 4 (2009), 15–18.

[101] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. *EMF: Eclipse Modeling Framework*, 2nd ed. Addison-Wesley Professional, 2008.

[102] Sulzmann, M. A General Type Inference Framework for Hindley/Milner Style Systems. In *FLOPS* (2001), H. Kuchen and K. Ueda, Eds., vol. 2024 of *LNCS*, Springer, pp. 248–263.

[103] Swierstra, S. D., Alcocer, P. R. A., and Saraiva, J. Designing and Implementing Combinator Languages. In *AFP* (1998), pp. 150–206.

[104] Tausch, N., Philippsen, M., and Adersberger, J. TracQL: A Domain-Specific Language for Traceability Analysis. In *WICSA/ECSA* (2012), IEEE, pp. 320–324.

[105] Tofte, M. *Operational semantics and polymorphic type inference*. PhD thesis, University of Edinburgh, 1988.

[106] Turing, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society s2-42*, 1 (1937), 230–265.

[107] Uhl, J., Drossopoulou, S., Persch, G., Goos, G., Dausmann, M., Winterstein, G., and Kirchgässner, W. *An Attribute Grammar for the Semantic Analysis of Ada*, vol. 139 of *LNCS*. Springer, 1982.

[108] Van Wyk, E., Bodin, D., Gao, J., and Krishnan, L. Silver: an Extensible Attribute Grammar System. *SCP 75*, 1–2 (2010), 39–54.

[109] Veerman, N. P. Revitalizing Modifiability of Legacy Assets. In *CSMR* (2003), IEEE Computer Society, pp. 19–29.

[110] Visser, E. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In *RTA* (2001), A. Middeldorp, Ed., vol. 2051 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 357–361.

[111] Visser, E. WebDSL: A Case Study in Domain-Specific Language Engineering. In *GTTSE* (2007), R. Lämmel, J. Visser, and J. Saraiva, Eds., vol. 5235 of *Lecture Notes in Computer Science*, Springer, pp. 291–373.

[112] Völter, M. Xtext/TS - a typesystem framework for Xtext. *InfoQ* (2011).

[113] Wilhelm, R., and Maurer, D. *Compiler design.* Addison-Wesley, 1995.

[114] Zeidler, U. ATLflow, 2011. http://opensource.urszeidler.de/ATLflow/.

Appendix A

Type system for SLCO

Listing A.1: SLCO EMF-TL Rules

```
imports
    http://www.emftext.org/language/textualSLCO;
    http://mdse.tue.nl/slco;
start textualSLCO::Model

typesystem
slcotype(type) =
  slco::PrimitiveType(primitiveTypeEnum);
widening

rules

from textualSLCO::StringConstantExpression
to slco::StringConstantExpression(type = $t)
where $t = slcotype{{type =
  slco::PrimitiveTypeEnum::String}}

from textualSLCO::BooleanConstantExpression
to slco::BooleanConstantExpression(type = $t)
where $t = slcotype{{type =
  slco::PrimitiveTypeEnum::Boolean}}

from textualSLCO::IntegerConstantExpression
to slco::IntegerConstantExpression(type = $t)
where $t = slcotype{{type =
  slco::PrimitiveTypeEnum::Integer}}

from
  textualSLCO::BinaryOperatorExpression(operator
  = $o, operand1 = $l, operand2 = $r)
to slco::BinaryOperatorExpression(type = $t)
where $o in set{textualSLCO::Operator::differs,
  textualSLCO::Operator::equals},
      $r.type = $l.type,
      $t = slcotype{{type =
        slco::PrimitiveTypeEnum::Boolean}}

from
  textualSLCO::BinaryOperatorExpression(operator
  = $o, operand1 = $l, operand2 = $r)
to slco::BinaryOperatorExpression(type = $t)
where $o in set{textualSLCO::Operator::add,
                textualSLCO::Operator::subtract},
      $l.type = $t,
      $r.type = $t,
      $t = slcotype{{type =
        slco::PrimitiveTypeEnum::Integer}}

from
  textualSLCO::BinaryOperatorExpression(operator
  = $o, operand1 = $l, operand2 = $r)
to slco::BinaryOperatorExpression(type = $t)
where $o = textualSLCO::Operator::and,
      $l.type = $t,
      $r.type = $t,
      $t = slcotype{{type =
        slco::PrimitiveTypeEnum::Boolean}}

from
  textualSLCO::BinaryOperatorExpression(operator
  = $o, operand1 = $l, operand2 = $r)
to slco::BinaryOperatorExpression(type = $t)
where $o = textualSLCO::Operator::atLeast,
      $l.type = slcotype{{type =
        slco::PrimitiveTypeEnum::Integer}},
      $r.type = slcotype{{type =
        slco::PrimitiveTypeEnum::Integer}},
      $t = slcotype{{type =
        slco::PrimitiveTypeEnum::Boolean}}

from textualSLCO::VariableExpression(links =
  $scope, name = $varname)
to slco::VariableExpression(type = $t, variable =
  $v)
where $v in $scope,
      $v = textualSLCO::Variable(name = $varname,
        type = $t)

from textualSLCO::Transition(sourcename =
  $source, targetname = $target, guard = $g,
  links = $links)
to slco::Transition(source = $sourcevertex,
  target = $targetvertex)
where $sourcevertex in $links,
      $sourcevertex = textualSLCO::Initial(name =
      $source),
      $targetvertex in $links,
      $targetvertex = textualSLCO::Initial(name =
        $target),
      $g.type = slcotype{{type =
        slco::PrimitiveTypeEnum::Boolean}}

from textualSLCO::AssignmentStatement(expression
  = $e,links = $links)
with $t
to slco::AssignmentStatement(variable = $variable)
where $variable in $links,
      $variable = textualSLCO::Variable(type =
      $t),
      $e.type = $t

from textualSLCO::Object(links = $links)
to slco::Object(class = $class)
where $class in $links,
      $class = textualSLCO::Class

from textualSLCO::SignalReception(links = $links)
to slco::SignalReception(port = $port)
where $port in $links,
      $port = textualSLCO::Port


from textualSLCO::SignalArgumentVariable(links =
  $scope)
to slco::SignalArgumentVariable(variable = $v)
where $v in $scope,
      $v = textualSLCO::Variable


from textualSLCO::SendSignalStatement(links =
  $links)
```

```
to slco::SendSignalStatement(port = $port)
where $port in $links,
      $port = textualSLCO::Port

from textualSLCO::Channel(links = $links, object1name = $object1name, object2name = $object2name, port1name = $port1name, port2name = $port2name)
to slco::Channel(object1 = $object1, object2 = $object2, port1 = $port1, port2 = $port2)
where $object1 in $links,
      $object1 = textualSLCO::Object(name = $object1name),
      $object2 in $links,
      $object2 = textualSLCO::Object(name = $object2name),
      $port1 in $links,
      $port1 = textualSLCO::Port(name = $port1name),
      $port2 in $links,
      $port2 = textualSLCO::Port(name = $port2name)
strategy

strategytarget
```

Appendix B

Type system for CIF

Listing B.1: CIF EMF-TL Rules rule

```
imports

 http://ucif.tue.nl/ucif-1.0.0;
 http://cif.tue.nl/cif-2.1.1;

start

    ucif::Specification

typesystem

bool = cif::types::BoolType;
real =cif::types::RealType;
int = cif::types::IntType;
nat = cif::types::NatType;
string = cif::types::StringType;

matrix(x, y, e) = cif::types::MatrixType
  (rowDimension,columnDimension,elementType);
array(d, e) =
  cif::types::ArrayType(dimension,elementType);
vector(d, e) =
  cif::types::VectorType(dimension,elementType);
list(e) = cif::types::ListType(elementType);
cifset(e) = cif::types::SetType(elementType);
function(p,r) =
  cif::types::FunctionType(parameterTypes,
  returnType);
distributiontype(r) =
  cif::types::DistributionType(resultType);
dictionary(k,v) =
  cif::types::DictionaryType(keyType,valueType);
tuple(f) = cif::types::TupleType(fields);

widening

nat< int
int< real

list(e=$e1) < list(e=$e2) if $e1 < $e2
array(d=$d1,e=$e1) < array(d=$d2, e=$e2) if $d1 =
  $d2 , $e1 < $e2


rules

from
  ucif::expressions::ArrayExpression(elements=$e)
with $te
to cif::expressions::ArrayExpression(type=$t)
where for $es in $e : $te = $es.type,
      $t = array{{d =
        cif::expressions::Number(value = length
        $e, type = nat{{}}), e = $te}}

from
  ucif::expressions::BinaryExpression(operator=$o,
  leftChild=$l, rightChild=$r)
with $tl,$tr
to cif::expressions::BinaryExpression(type=$t)
where $o in
  set{ucif::expressions::BinaryOperators::Addition,
  ucif::expressions::BinaryOperators::Multiplication,
  ucif::expressions::BinaryOperators::Subtraction,
  ucif::expressions::BinaryOperators::Minimum},
     $t < real{{}},
     $tl = $l.type,
     $tl < real{{}},
     $tl < $t,
     $tr = $r.type,
     $tr < real{{}},
     $tr < $t

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
```

```
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Concatenation,
     $t = $l.type,
     $t = $r.type,
     $t< string{{}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $te
to cif::expressions::BinaryExpression(type=$t)
where $o in
  set{ucif::expressions::BinaryOperators::Concatenation,
  ucif::expressions::BinaryOperators::ListSubtraction},
     $t = $l.type,
     $t = $r.type,
     $t = list{{e = $te}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $tr,$tl,$dl,$dr,$te,$dres
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Concatenation,
     $tl = $l.type,
     $tr = $r.type,
     $tl = array{{d =
       cif::expressions::Number(value = $dl), e =
       $te}},
     $tr = array{{d =
       cif::expressions::Number(value = $dr), e =
       $te}},
     $dres = cif::expressions::Number(value = $dl
       + $dr,type = nat{{}}),
     $t = array{{d = $dres, e = $te}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $tr,$tl,$dl,$dr,$te, $dres
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Concatenation,
     $tl = $l.type,
     $tr = $r.type,
     $tl = vector{{d =
       cif::expressions::Number(value = $dl), e =
       $te}},
     $tr = vector{{d =
       cif::expressions::Number(value = $dr), e =
       $te}},
     $dres = cif::expressions::Number(value = $dl
       + $dr,type = nat{{}}),
     $t = vector{{d = $dres, e = $te}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
to cif::expressions::BinaryExpression(type=$t)
where $o in set{
  ucif::expressions::BinaryOperators::ConditionalConjunction,
  ucif::expressions::BinaryOperators::ConditionalDisjunction,
  ucif::expressions::BinaryOperators::Conjunction,
  ucif::expressions::BinaryOperators::Disjunction,
  ucif::expressions::BinaryOperators::Implication},
     $t = $l.type,
     $t = $r.type,
     $t = bool{{}}

from
  ucif::expressions::BinaryExpression(operator=$o,
  leftChild=$l, rightChild=$r)
with $tl,$tr
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Division,
     $t = real{{}},
     $tl = $l.type,
     $tl < real{{}},
     $tr = $r.type,
     $tr < real{{}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $tl, $tr
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::ElementTest,
     $tl = $l.type,
     $tr = $r.type,
     $tr = list{{e = $tl}},
     $t = bool{{}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $tl, $tr
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::ElementTest,
     $tl = $l.type,
     $tr = $r.type,
     $tr = cifset{{e = $tl}},
     $t = bool{{}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $u
to cif::expressions::BinaryExpression(type=$t)
where $o in
  set{ucif::expressions::BinaryOperators::Equal,
  ucif::expressions::BinaryOperators::NotEqual},
     $u = $l.type,
     $u = $r.type,
     $t = bool{{}}

from
  ucif::expressions::BinaryExpression(operator=$o,
  leftChild=$l, rightChild=$r)
with $tl,$tr
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::FloorDivision,
     $t = nat{{}},
     $tl = $l.type,
     $tl =nat{{}},
     $tr = $r.type,
     $tr = nat{{}}

from
  ucif::expressions::BinaryExpression(operator=$o,
  leftChild=$l, rightChild=$r)
with $tl,$tr
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::FloorDivision,
     $t = int{{}},
     $tl = $l.type,
     $tl < real{{}},
     $tr = $r.type,
     $tr < real{{}}
```

```

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $u
to cif::expressions::BinaryExpression(type=$t)
where $o in set{
  ucif::expressions::BinaryOperators::GreaterEqual,
  ucif::expressions::BinaryOperators::GreaterThan,
  ucif::expressions::BinaryOperators::LessEqual,
  ucif::expressions::BinaryOperators::LessThan},
    $u = $l.type,
    $u = $r.type,
    $u < real{{}},
    $t = bool{{}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $u
to cif::expressions::BinaryExpression(type=$t)
where $o in
  set{ucif::expressions::BinaryOperators::GreaterEqual,
  ucif::expressions::BinaryOperators::GreaterThan,
  ucif::expressions::BinaryOperators::LessEqual,
  ucif::expressions::BinaryOperators::LessThan},
    $u = $l.type,
    $u = $r.type,
    $u = string{{}},
    $t = bool{{}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $te
to cif::expressions::BinaryExpression(type=$t)
where $o in
  set{ucif::expressions::BinaryOperators::Intersection,
  ucif::expressions::BinaryOperators::SetSubtraction,
  ucif::expressions::BinaryOperators::Union},
    $t = $l.type,
    $t = $r.type,
    $t = cifset{{e = $te}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Maximum,
      $t = $l.type,
      $t = nat{{}},
      $t = $r.type

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $tr
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Maximum,
      $t = $l.type,
      $t = nat{{}},
      $tr = $r.type,
      $tr = int{{}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $tl
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Maximum,
      $tl = $l.type,
      $tl = int{{}},
      $t = $r.type,
      $t = nat{{}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Maximum,
      $t = $l.type,
      $t = int{{}},
      $t = $r.type

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Maximum,
      $t = $l.type,
      $t = real{{}},
      $t = $r.type

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
```

```
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Maximum,
      $t = $l.type,
      $t = string{{}},
      $t = $r.type

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Modulus,
      $t = $l.type,
      $t < int{{}},
      $t = $r.type

from
  ucif::expressions::BinaryExpression(operator=$o,
  leftChild=$l, rightChild=$r)
with $tl,$tr
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Power,
     $t = nat{{}},
     $tl = $l.type,
     $tl = nat{{}},
     $tr = $r.type,
     $tr = nat{{}}

from
  ucif::expressions::BinaryExpression(operator=$o,
  leftChild=$l, rightChild=$r)
with $tl,$tr
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Power,
     $t = int{{}},
     $tl = $l.type,
     $tl = int{{}},
     $tr = $r.type,
     $tr = nat{{}}

from
  ucif::expressions::BinaryExpression(operator=$o,
  leftChild=$l, rightChild=$r)
with $tl,$tr
```

```
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Power,
     $t = real{{}},
     $tl = $l.type,
     $tl = int{{}},
     $tr = $r.type,
     $tr = real{{}}

from
  ucif::expressions::BinaryExpression(operator=$o,
  leftChild=$l, rightChild=$r)
with $tl,$tr
to cif::expressions::BinaryExpression(type=$t)
where $o =
  ucif::expressions::BinaryOperators::Power,
     $t = real{{}},
     $tl = $l.type,
     $tl = int{{}},
     $tr = $r.type,
     $tr > int{{}},
     $tr < real{{}}

from ucif::expressions::BinaryExpression(operator
  = $o, leftChild=$l, rightChild=$r)
with $ts,$te
to cif::expressions::BinaryExpression(type=$t)
where $o in
  set{ucif::expressions::BinaryOperators::Subset},
     $ts = $l.type,
     $ts = $r.type,
     $ts = cifset{{e = $te}},
     $t = bool{{}}

from ucif::expressions::BoolLiteral
to cif::expressions::BoolLiteral(type=$t)
where $t= bool{{}}

from
  ucif::expressions::ReferenceExpression(links=$scope,
  text = $name)
to cif::expressions::ClockReference(type=$t,
  clock=$o)
where $o in $scope,
      $o = cif::Clock(staticType=$t, name = $name)

```

```
from
  ucif::expressions::ConditionalExpression(alternatives=$a)
with $guards,$values, $tg
to
  cif::expressions::ConditionalExpression(type=$t)
where for $alt in $a, $guard in $guards, $exp in
  $values : $alt =
  ucif::expressions::ConditionalAlternative(guard
  = $guard, value = $exp),
      for $guard in $guards : $tg = $guard.type,
      $tg = bool{{}},
      for $exp in $values : $t = $exp.type

from
  ucif::expressions::DictionaryExpression(pairs=
  $p)
with $tkey, $keys, $tvalue, $values
to cif::expressions::DictionaryExpression(type =
  $t)
where for $pair in $p , $key in $keys, $value in
  $values: $pair =
  ucif::expressions::DictionaryPair(key=$key,value=$value),
      for $key in $keys : $tkey = $key.type,
      for $value in $values : $tvalue =
        $value.type,
      $t = dictionary{{k=$tkey,v=$tvalue}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Constant",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $ptypes = set{bool{{}}},
      $te = $s.type,
      $te = nat{{}},
      $t = bool{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes
to cif::expressions::Distribution(type=$t)
where $n = "Constant",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $ptypes = set{bool{{}}},
      $s = OclUndefined,
      $t = bool{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Constant",
      for $ptype in $ptypes, $p in $ps : $ptype
       =$p.type,
      $ptypes = set{nat{{}}},
      $te = $s.type,
      $te = nat{{}},
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes
to cif::expressions::Distribution(type=$t)
where $n = "Constant",
      for $ptype in $ptypes, $p in $ps : $ptype
       =$p.type,
      $ptypes = set{nat{{}}},
      $s = OclUndefined,
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Constant",
      for $ptype in $ptypes, $p in $ps : $ptype
       =$p.type,
      $argtypes = set{int{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = int{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Constant",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{int{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = int{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Constant",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Constant",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Bernoulli",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = bool{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Bernoulli",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = bool{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Bernoulli",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Bernoulli",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
```

```
to cif::expressions::Distribution(type=$t)
where $n = "Beta",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Beta",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Binomial",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},nat{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Binomial",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},nat{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Erlang",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{nat{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Erlang",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{nat{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Exponential",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

```

```
from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Exponential",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Gamma",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Gamma",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Geometric",
      for $ptype in $ptypes, $p in $ps : $ptype
```

```
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Geometric",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "LogNormal",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "LogNormal",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Normal",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Normal",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Poisson",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
```

```
to cif::expressions::Distribution(type=$t)
where $n = "Poisson",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te, $length
to cif::expressions::Distribution(type=$t)
where $n = "Random",
      $length = length $ps,
      $length = 0,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $length
to cif::expressions::Distribution(type=$t)
where $n = "Random",
      $length = length $ps,
      $length = 0,
      $s = OclUndefined,
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Triangle",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Triangle",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Uniform",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{nat{{}},nat{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Uniform",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{nat{{}},nat{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = nat{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Uniform",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{int{{}},int{{}}},
      for $ptype in $ptypes, $argtype in $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = int{{}}

from ucif::expressions::Distribution(name = $n, parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Uniform",
      for $ptype in $ptypes, $p in $ps : $ptype =$p.type,
      $argtypes = set{int{{}},int{{}}},
      for $ptype in $ptypes, $argtype in $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = int{{}}

from ucif::expressions::Distribution(name = $n, parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Uniform",
      for $ptype in $ptypes, $p in $ps : $ptype =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

from ucif::expressions::Distribution(name = $n, parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type=$t)
where $n = "Uniform",
      for $ptype in $ptypes, $p in $ps : $ptype =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = real{{}}
```

```

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes, $te
to cif::expressions::Distribution(type=$t)
where $n = "Weibull",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $te = $s.type,
      $te = nat{{}},
      $t = real{{}}

from ucif::expressions::Distribution(name = $n,
  parameters = $ps, seed = $s)
with $ptypes, $argtypes
to cif::expressions::Distribution(type = $t)
where $n = "Weibull",
      for $ptype in $ptypes, $p in $ps : $ptype
        =$p.type,
      $argtypes = set{real{{}},real{{}}},
      for $ptype in $ptypes, $argtype in
        $argtypes : $ptype < $argtype,
      $s = OclUndefined,
      $t = real{{}}

from ucif::expressions::ReferenceExpression(links
  = $scope,text = $name)
to cif::expressions::FieldReference(type = $t,
  field = $o)
where $o in $scope,
      $o = cif::types::Field(name = $name, type =
        $t)

from ucif::expressions::FunctionCallExpression(
  function = $function, arguments = $arguments)
with $functiontype, $parametertypes
to cif::expressions::FunctionCallExpression( type
  = $returntype)
where $functiontype = $function.type,
      $functiontype = function{{p =
        $parametertypes, r = $returntype}},
      for $pt in $parametertypes, $a in
        $arguments : $pt = $a.type
```

```

from
  ucif::expressions::ReferenceExpression(links=$scope,
  text = $name)
with $parameters,$parametertypes,$returntype
to cif::expressions::FunctionReference(type=$t,
  function=$function)
where $function in $scope,
      $function =
        cif::InternalFunctionDeclaration(name =
        $name, formalParameters=$parameters,
        returnType=$returntype),
      for $p in $parameters, $pt in
        $parametertypes : $p =
        cif::Variable(staticType=$pt),
      $t = function{{p = $parametertypes, r =
        $returntype}}

from
  ucif::expressions::ReferenceExpression(links=$scope,
  text = $name)
to
  cif::expressions::GlobalConstantReference(type=$t,
  constant=$o)
where $o in $scope,
      $o = cif::ConstantDeclaration(type=t, name
        = $name)

from
  ucif::expressions::LambdaExpression(formalParameters
  = $fps, returnExpression = $exp, returnType =
  $rt)
with $ptypes
to cif::expressions::LambdaExpression(type = $t)
where for $ptype in $ptypes, $fp in $fps : $fp =
  ucif::expressions::Parameter(type=$ptype),
     $rt > $exp.type,
     $t = function{{p = $ptypes, r = $rt}}

from
  ucif::expressions::ListExpression(elements=$e)
with $te
to cif::expressions::ListExpression(type=$t)
where for $es in $e : $te = $es.type,
      $t = list{{e = $te}}

```

```
from
  ucif::expressions::ReferenceExpression(links=$scope,
  text = $name)
to
  cif::expressions::LocalConstantReference(type=$t,
  constant=$o)
where $o in $scope,
      $o = cif::Constant(staticType=$t, name =
        $name)

from
  ucif::expressions::MatrixExpression(rows=$mrs)
with $rows, $nc, $tr
to cif::expressions::MatrixExpression(type=$t)
where for $row in $rows, $mr in $mrs : $mr =
  ucif::expressions::MatrixRow(columnElements=$row),
      for $row in $rows : ($nc = length $row,for
        $elem in $row : $tr = $elem.type),
      $tr = real{{}},
      $t = matrix{{x = $nc, y = length $rows, e =
        real{{}}}}

from ucif::expressions::Number
to cif::expressions::Number(type=$t)
where $t< real{{}}

from
  ucif::expressions::ReferenceExpression(links=$scope,
  text = $name)
to cif::expressions::ParameterReference(type=$t,
  parameter=$o)
where $o in $scope,
      $o = cif::expressions::Parameter(type=$t,
        name = $name)

from ucif::expressions::RealNumber
to cif::expressions::RealNumber(type=$t)
where $t = real{{}}

from ucif::expressions::SetExpression(elements=$e)
with $te
to cif::expressions::SetExpression(type=$t)
where for $es in $e : $te = $es.type,
      $t = cifset{{e = $te}}

from
```

```
  ucif::expressions::ReferenceExpression(links=$scope,
  text = $name)
with $parameters,$parametertypes,$returntype
to
  cif::expressions::StdLibFunctionReference(type=$t,
  function=$function)
where $function in $scope,
      $function =
        cif::InternalFunctionDeclaration (name =
        $name, formalParameters=$parameters,
        returnType=$returntype),
      for $p in $parameters, $pt in
        $parametertypes : $p =
        cif::Variable(staticType=$pt),
      $t = function{{p = $parametertypes, r =
        $returntype}}

from ucif::expressions::StringLiteral
to cif::expressions::StringLiteral(type=$t)
where $t= string{{}}

from ucif::expressions::TimeLiteral
to cif::expressions::TimeLiteral(type=$t)
where $t = real{{}}

from ucif::expressions::TupleExpression(fields =
  $fields)
with $tfields
to cif::expressions::TupleExpression(type=$t)
where for $tfield in $tfields, $field in $fields:
  $tfield = cif::types::Field(type=$field.type),
      $t = tuple{{f=$tfields}}

from ucif::expressions::UnaryExpression(operator
  = $o, child = $c)
with $v
to cif::expressions::UnaryExpression(type=$t)
where $o in
  set{ucif::expressions::UnaryOperators::Derivative,
  ucif::expressions::UnaryOperators::New},
     $v = $c.variable,
     $v = cif::Variable,
     $t = $c.type,
     $t< real{{}}

from ucif::expressions::UnaryExpression(operator
  = $o, child = $c)
with $v
to cif::expressions::UnaryExpression(type=$t)
where $o in
  set{ucif::expressions::UnaryOperators::Derivative,
  ucif::expressions::UnaryOperators::New},
     $v = $c.clock,
     $v = cif::Clock,
     $t = $c.type,
     $t< real{{}}

from ucif::expressions::UnaryExpression(operator
  = $o, child = $c)
to cif::expressions::UnaryExpression(type=$t)
where $o =
  ucif::expressions::UnaryOperators::Inverse,
     $t = $c.type,
     $t < int{{}}

from ucif::expressions::UnaryExpression(operator
  = $o, child = $c)
to cif::expressions::UnaryExpression(type=$t)
where $o =
  ucif::expressions::UnaryOperators::Inverse,
     $t = $c.type,
     $t = real{{}}

from ucif::expressions::UnaryExpression(operator
  = $o, child = $c)
to cif::expressions::UnaryExpression(type=$t)
where $o =
  ucif::expressions::UnaryOperators::Negate,
     $t = $c.type,
     $t = bool{{}}

from ucif::expressions::UnaryExpression(operator
  = $o, child = $c)
with $tc
to cif::expressions::UnaryExpression(type=$t)
where $o =
  ucif::expressions::UnaryOperators::Pick,
     $tc = $c.type,
     $tc = cifset{{e=$t}}

from ucif::expressions::UnaryExpression(operator
  = $o, child = $c)
```

```
with $tc
to cif::expressions::UnaryExpression(type=$t)
where $o =
  ucif::expressions::UnaryOperators::Sample,
     $tc = $c.type,
     $tc = distributiontype{{r=$t}}

from
  ucif::expressions::ReferenceExpression(links=$scope)
to cif::expressions::VariableReference(type=$t,
  variable=$o)
where $o in $scope,
      $o = cif::Variable(staticType=$t)

from
  ucif::expressions::VectorExpression(elements=$e)
with $te
to cif::expressions::VectorExpression(type=$t)
where for $es in $e : $te = $es.type,
      $te < real{{}},
      $t = vector{{d =
        cif::expressions::Number(value = length
        $e, type = nat{{}}), e = $te}}


strategy
nat< int
int< real
list(e=$e1) < list(e=$e2) if $e1 < $e2
array(d=$d1,e=$e1) < array(d=$d2, e=$e2) if $d1 =
  $d2 , $e1 < $e2

strategytarget
  type
```

Appendix C

Type system for WebDSL

Listing C.1: WebDSL EMF-TL Rules

```
imports
    http://mdse.tue.nl/textualWebDSL.ecore;
    http://mdse.tue.nl/WebDSL.ecore;

start textualWebDSL::Unit

typesystem
  simpletype(sort,constraintvar) <
    textualWebDSL::SimpleSort(name,constraintvar)>
    WebDSL::SimpleSort(name);
  generictype(sort,args) =
    WebDSL::GenericSort(name,sorts);

widening

  simpletype(sort = $s1 ) < simpletype(sort =
    $s2) if $s1 = "Integer", $s2 = "Float"

rules
  from textualWebDSL::Int
  to WebDSL::Int(resultSort=$t)
  where $t = simpletype{{sort = "Integer"}}

  from textualWebDSL::Float
  to WebDSL::Float(resultSort=$t)
  where $t = simpletype{{sort = "Float"}}

  from textualWebDSL::BooleanLiteral
  to WebDSL::BooleanLiteral(resultSort=$t)
```

```
where $t = simpletype{{sort = "Boolean"}}

from textualWebDSL::StringLiteral
to WebDSL::String(resultSort=$t)
where $t = simpletype{{sort = "String"}}

from textualWebDSL::BinaryExpression(operator =
  $o,
                                     left=$l,
                                     right = $r)
to WebDSL::BinaryExpression(resultSort=$t)
where $o in
  set{textualWebDSL::BinaryOperator::Add,
  textualWebDSL::BinaryOperator::Sub,
  textualWebDSL::BinaryOperator::Mul},
      $t > $l.resultSort,
      $t > $r.resultSort,
      $t < simpletype{{sort = "Float"}}

error textualWebDSL::BinaryExpression(operator
  = $o,position = $pos)
message "Only numbers can be added, subtracted
  or multiplied"
sourceposition $pos
where $o in
  set{textualWebDSL::BinaryOperator::Add,
  textualWebDSL::BinaryOperator::Sub,
  textualWebDSL::BinaryOperator::Mul}

from textualWebDSL::BinaryExpression(operator =
  $o,
                                     left=$l,
                                     right = $r)
with $te
to WebDSL::BinaryExpression(resultSort=$t)
where $o in
  set{textualWebDSL::BinaryOperator::LargerThan,
  textualWebDSL::BinaryOperator::LargerThanOrEqual,
  textualWebDSL::BinaryOperator::SmallerThan,
  textualWebDSL::BinaryOperator::SmallerThanOrEqual},
      $te > $l.resultSort,
      $te > $r.resultSort,
      $te < simpletype{{sort = "Float"}},
      $t = simpletype{{sort = "Boolean"}}

error textualWebDSL::BinaryExpression(operator
  = $o,position = $pos)
message "Only numbers can be compared based on
  size"
sourceposition $pos
where $o in
  set{textualWebDSL::BinaryOperator::LargerThan,
  textualWebDSL::BinaryOperator::LargerThanOrEqual,
  textualWebDSL::BinaryOperator::SmallerThan,
  textualWebDSL::BinaryOperator::SmallerThanOrEqual}

from textualWebDSL::BinaryExpression(operator =
  $o,
                                      left=$l,
                                      right = $r)
with $te
to WebDSL::BinaryExpression(resultSort=$t)
where $o in
  set{textualWebDSL::BinaryOperator::Eq,
  textualWebDSL::BinaryOperator::NotEq},
      $te > $l.resultSort,
      $te > $r.resultSort,
      $t = simpletype{{sort = "Boolean"}}

error textualWebDSL::BinaryExpression(operator
  = $o,position = $pos)
message "Only values of similar types can be
  compared"
sourceposition $pos
where $o in
  set{textualWebDSL::BinaryOperator::Eq,
  textualWebDSL::BinaryOperator::NotEq}

from textualWebDSL::Cast(exp = $exp, sort =
  $ttarget)
to WebDSL::Cast(resultSort = $ttarget)

from textualWebDSL::IsA
to WebDSL::Cast(resultSort = $t)
where $t = simpletype{{sort = "Boolean"}}

from textualWebDSL::SetCreation(elements =
  $elems, sort = $telem)
to WebDSL::SetCreation(resultSort = $t)
where for $e in $elems : $telem = $e.resultSort,
      $t = generictype{{sort="Set",
        args=set{$telem} }}
```

```

error textualWebDSL::SetCreation(position = $pos)
message "All elements of a set must have the same type"
sourceposition $pos

from textualWebDSL::ListCreation(elements = $elems, sort = $telem)
to WebDSL::ListCreation(resultSort = $t)
where for $e in $elems : $telem = $e.resultSort,
      $t = generictype{{sort="Set", args=set{$telem} }}

error textualWebDSL::ListCreation(position = $pos)
message "All elements of a list must have the same type"
sourceposition $pos

from textualWebDSL::MapCreation(mapping=$mapping)
with $keys, $tkey, $values, $tvalue
to WebDSL::MapCreation(resultSort = $t)
where for $m in $mapping, $k in $keys, $v in $values : $m = textualWebDSL::Mapping(key = $k, value = $v),
      for $k1 in $keys : $tkey = $k1.resultSort,
      for $v1 in $values : $tvalue > $v1.resultSort,
      $t = generictype{{sort="Map", args=set{$tkey, $tvalue} }}

error textualWebDSL::MapCreation(mapping=$mapping,position = $pos)
message "All keys of a map must have similar types"
sourceposition $pos
with $values, $tvalue
where for $m in $mapping, $v in $values : $m = textualWebDSL::Mapping(value = $v),
      for $v in $values : $tvalue > $v.resultSort

error textualWebDSL::MapCreation(mapping=$mapping,position = $pos)
```

```
message "All keys of a map must have similar
  types"
sourceposition $pos
with $keys, $tkey
where for $m in $mapping, $k in $keys : $m =
  textualWebDSL::Mapping(key = $k),
      for $k in $keys : $tkey = $k1.resultSort

from textualWebDSL::VariableAccess(links =
  $links, variable = $name)
to WebDSL::VariableAccess(variable = $var,
  resultSort = $t)
where $var in $links,
      $var =
        textualWebDSL::AbstractVarDecl(name =
        $name,sort = $t)

error textualWebDSL::VariableAccess(links =
  $links, variable = $name, position = $pos)
message "No variable with given name in scope"
sourceposition $pos

from textualWebDSL::FieldAccess(links = $links,
  base = $exp, position = $pos, field = $name)
with $exptype, $entity, $props, $name
to WebDSL::FieldAccess(field = $field,
  resultSort = $t)
where $exptype = $exp.resultSort,
      $entity = $exptype.entity,
      $props = $entity.allBody,
      $field in $links,
      $field = textualWebDSL::Property(sort =
        $t, name = $name),
      $field in $props

error textualWebDSL::FieldAccess(base = $exp,
  position = $pos)
message "Only entities have fields"
sourceposition $pos
with $texp, $ent
where $texp = $exp.resultSort,
      $ent = $texp.entity,
      $ent = OclUndefined

error textualWebDSL::FieldAccess(base = $exp,
  position = $pos)
```

```
message "Entity does not have property with
  given name"
sourceposition $pos
with $texp, $ent, $props
where $texp = $exp.resultSort,
      $ent = $texp.entity,
      $props = $ent.allBody,
      $field in $links,
      $field = textualWebDSL::Property(sort =
        $t),
      for $p in $props : $p # $field

from textualWebDSL::ObjectCreation(sort = $s,
  objectPropertiesAssignments = $ps)
with $e, $b, $panames, $props
to WebDSL::ObjectCreation(resultSort = $s)
where $e = $s.entity,
      $b = $e.allBody,
      for $aname in $panames, $pa in $ps, $prop
        in $props :
         ($pa =
           textualWebDSL::ObjectPropertyAssignment
           (property = $aname),
          $prop = textualWebDSL::Property (name
            = $aname),
          $prop in $b)

error textualWebDSL::ObjectCreation(sort = $s,
  position = $pos)
message "Only objects can be created using this
  operator"
sourceposition $pos
with $e
where $e = $s.entity,
      $e = OclUndefined

error textualWebDSL::ObjectCreation(sort = $s,
  objectPropertiesAssignments = $ps)
message "Some properties do not exist in target
  object"
sourceposition $pos
with $e, $b, $pa
where $e = $s.entity,
      $b = $e.allBody,
      $pa in $ps,
      $pa =
        textualWebDSL::ObjectPropertyAssignment(property
        = $aname),
      for $prop in b : ($prop =
        textualWebDSL::Property, $aname #
        $p.name)

from textualWebDSL::ObjectPropertyAssignment(
  links = $links, value = $val)
with $t
to WebDSL::ObjectPropertyAssignment(property =
  $property)
where $property in $links,
      $property = textualWebDSL::Property(sort
        = $t),
      $t = $val.resultSort

error textualWebDSL::ObjectPropertyAssignment
message "No such property exists"

from textualWebDSL::Call(base = $base, links =
  $links, arguments = $args)
with $fparams, $fparamtypes
to WebDSL::Call(function = $f, resultSort = $t)
where $base = OclUndefined,
      $f in $links,
      $f = textualWebDSL::Function(arguments =
        $fparams, returnSort=$t),
      for $fparam in $fparams, $fparamtype in
        $fparamtypes : $fparam =
        textualWebDSL::FormalArg(sort =
        $fparamtype),
      for $arg in $args, $ftype in $fparamtypes
        : $ftype > $arg.resultSort

error textualWebDSL::Call(base = $base,
  position = $pos)
message "Function does not exist"
sourceposition $pos
where $base = OclUndefined,
      for $f in $links : $f #
        textualWebDSL::Function(arguments =
        $fparams, returnSort=$t)

error textualWebDSL::Call(base = $base,
  position = $pos)
message "Types of arguments do not match
```

```
  expected parameters"
sourceposition $pos
with $f
where $base = OclUndefined,
      $f in $links,
      $f = textualWebDSL::Function

from textualWebDSL::Call(base = $base, links =
  $links, arguments = $args)
with $texp, $ent, $body, $fparams, $fparamtypes
to WebDSL::Call(function = $f, resultSort = $t)
where $texp = $base.resultSort,
      $ent = $texp.entity,
      $ent = textualWebDSL::Entity,
      $body = $ent.allBody,
      $f in $body,
      $f in $links,
      $f = textualWebDSL::Function(arguments =
        $fparams, returnSort=$t),
      for $fparam in $fparams, $fparamtype in
        $fparamtypes : $fparam =
        textualWebDSL::FormalArg(sort =
        $fparamtype),
      for $arg in $args, $ftype in $fparamtypes
        : $ftype > $arg.resultSort

error textualWebDSL::Call(base = $base, links =
  $links, arguments = $args, position = $pos)
message "Only objects have member functions"
sourceposition $pos
where $texp = $base.resultSort,
      $ent = $texp.entity,
      $ent # textualWebDSL::Entity

error textualWebDSL::Call(base = $base, links =
  $links, position = $pos)
message "Function does not exist in this object"
sourceposition $pos
with $texp, $ent, $body, $f
where $texp = $base.resultSort,
      $ent = $texp.entity,
      $ent = textualWebDSL::Entity,
      $body = $ent.allBody,
      $f in $body,
      for $l in $links : $f # $l

```

```
error textualWebDSL::Call(base = $base, links = $links, position = $pos)
message "Only functions can be called"
sourceposition $pos
with $texp, $ent, $body, $f
where $texp = $base.resultSort,
      $ent = $texp.entity,
      $ent = textualWebDSL::Entity,
      $body = $ent.allBody,
      $f in $body,
      $f in $links,
      $f # textualWebDSL::Function

error textualWebDSL::Call(base = $base, links = $links, position = $pos)
message "Types of arguments do not match expected parameters"
sourceposition $pos
with $texp, $ent, $body, $f
where $texp = $base.resultSort,
      $ent = $texp.entity,
      $ent = textualWebDSL::Entity,
      $body = $ent.allBody,
      $f in $body,
      $f in $links,
      $f # textualWebDSL::Function

from textualWebDSL::SimpleSort(name = $s)
to WebDSL::SimpleSort
where $s = "String"

from textualWebDSL::SimpleSort(name = $s)
to WebDSL::SimpleSort
where $s = "Integer"

from textualWebDSL::SimpleSort(name = $s)
to WebDSL::SimpleSort
where $s = "Float"

from textualWebDSL::SimpleSort(name = $s)
to WebDSL::SimpleSort
where $s = "Boolean"

from textualWebDSL::SimpleSort(links = $links,name = $s)
to WebDSL::SimpleSort(entity = $e)
```

```
  where $e in $links,
        $e = WebDSL::Entity(name = $s)

  error textualWebDSL::SimpleSort
  message "No such entity exists"

  from textualWebDSL::Entity(body = $body, links = $links, superentity = $s)
  with $superb
  to WebDSL::Entity(superentity = $e, allBody = $allbody)
  where $e in $links,
        $e = WebDSL::Entity(name = $s),
        $superb = $e.allBody,
        $allbody = $superb & $body

  error textualWebDSL::Entity(body = $body, links = $links, superentity = $s)
  message "No such entity exists"
  where for $e in $links : $e # WebDSL::Entity(name = $s)

  from textualWebDSL::GenericSort
  to WebDSL::GenericSort



  from textualWebDSL::Entity(body = $b, superentity = $s)
  to WebDSL::Entity(allBody = $b)
  where $s = OclUndefined

strategy
  simpletype(sort = $s1) < simpletype(sort = $s2) if $s1 = "Integer", $s2 = "Float"
WebDSL::Function(name = $n1, arguments = $args1) < WebDSL::Function(name = $n2, arguments = $args2)
      if $n1 = $n2,
         for $arg1 in $args1,
         $arg2 in $args2 : $arg1 < $arg2
WebDSL::FormalArg(sort = $s1) < WebDSL::FormalArg(sort = $s2)
      if $s1 < $s2
```

```
strategytarget
  resultSort function
```

Appendix D

Type system for mCRL2

Listing D.1: mCRL2 EMF-TL Rules

```
imports http://mdse.tue.nl/mcrl2;
        http://mdse.tue.nl/textualmcrl2;

start textualmcrl2::Specification
typesystem
widening
   mcrl2::Pos < mcrl2::Nat
   mcrl2::Nat <  mcrl2::Int
   mcrl2::Int  < mcrl2::Real
   mcrl2::PosSort < mcrl2::NatSort
   mcrl2::NatSort <  mcrl2::IntSort
   mcrl2::IntSort  < mcrl2::RealSort
   mcrl2::ListSort(elementSort = $el) <
     mcrl2::ListSort(elementSort = $er) if $el <
     $er
   mcrl2::SetSort(elementSort = $el) <
     mcrl2::SetSort(elementSort = $er) if $el <
     $er
   mcrl2::BagSort(elementSort = $el) <
     mcrl2::BagSort(elementSort = $er) if $el <
     $er

rules
   from textualmcrl2::Bool
   to mcrl2::Bool(sort = $t)
   where $t = mcrl2::BoolSort

   from textualmcrl2::Pos
   to mcrl2::Pos(sort = $t)
```

```
where $t = mcrl2::PosSort

from textualmcrl2::Nat
to mcrl2::Nat(sort = $t)
where $t = mcrl2::NatSort

from textualmcrl2::Int
to mcrl2::Int(sort = $t)
where $t = mcrl2::IntSort

from textualmcrl2::Real
to mcrl2::Real(sort = $t)
where $t = mcrl2::RealSort

from textualmcrl2::List(elementSort=$es)
to mcrl2::List(sort = $t)
where $t =
  mcrl2::ListSort(elementSort=$es.sort)

from textualmcrl2::Set(elementSort=$es)
to mcrl2::Set(sort = $t)
where $t = mcrl2::SetSort(elementSort=$es.sort)

from textualmcrl2::Bag(elementSort=$es)
to mcrl2::Bag(sort = $t)
where $t = mcrl2::BagSort(elementSort=$es.sort)

from textualmcrl2::HigherOrder(domain =
  $dlist, result = $r)
with $dslist,$rsort
to mcrl2::HigherOrder(sort = $t)
where for $d in $dlist, $ds in $dslist : $ds =
  $d.sort,
      $rsort = $r.sort,
      $t = mcrl2::HigherOrderSort
               (domain = $dslist, result =
                 $rsort)

from textualmcrl2::SortRef(sortname=$n,links =
  $env)
to mcrl2::SortRef(sort=$s)
where $s in $env,
      $s = textualmcrl2::StructureSort(name =
        $n)

from textualmcrl2::SortRef(sortname=$n,links =
  $env)
with $es
to mcrl2::SortRef(sort=$s)
where $es in $env,
      $es = textualmcrl2::ExpressionSort(name
        = $n),
      $s = $es.expression.sort


error textualmcrl2::SortRef
message "No such sort"

from textualmcrl2::AtomicAction(atomname =
  $name, links = $env, arguments = $args)
with $t
to mcrl2::AtomicAction(atom = $a)
where $a in $env,
      $a = textualmcrl2::Atom(name = $name,
        type = $t),
      for $arg in $args,
          $param in $t :
          $arg.type.sort < $param.sort

from textualmcrl2::AtomicAction(atomname =
  $name, links = $env, arguments = $args)
with $t
to mcrl2::Instance(process = $p)
where $p in $env,
      $p = textualmcrl2::Process
              (name = $name, parameters = $t),
      for $arg in $args,
          $param in $t :
          $arg.type.sort < $param.sort.sort

from textualmcrl2::AtomicAction(atomname =
  $name, links = $env, arguments = $args) in
  $parent: textualmcrl2::ProcessDecl
to mcrl2::Instance(process = $p)
where length $args = 0,
      $parent =
        textualmcrl2::ProcessDecl(process =
        $p),
      $p = textualmcrl2::Process(name=$name)

from textualmcrl2::Block(atomnames = $anames,
  links = $env)
```

```
to mcrl2::Block(atoms = $atoms)
where for $aname in $anames,
          $atom in $atoms :
          ($atom in $env,
           $atom = textualmcrl2::Atom(name =
             $aname))

from textualmcrl2::Allow(atomnames = $anames,
  links = $env)
to mcrl2::Allow(atoms = $atoms)
where for $aname in $anames,
          $atom in $atoms :
          ($atom in $env,
           $atom = textualmcrl2::Atom(name =
             $aname))

from textualmcrl2::Hide(atomnames = $anames,
  links = $env)
to mcrl2::Hide(atoms = $atoms)
where for $aname in $anames,
          $atom in $atoms :
          ($atom in $env,
           $atom = textualmcrl2::Atom(name =
             $aname))

from textualmcrl2::Rename
to mcrl2::Rename

from textualmcrl2::Renaming(oldname = $o,
  newname = $n, links = $env)
to mcrl2::Renaming(old = $oatom, new = $natom)
where $oatom in $env,
      $oatom = textualmcrl2::Atom(name = $o),
      $natom in $env,
      $natom = textualmcrl2::Atom(name = $n)

from textualmcrl2::MultiAction(actionnames =
  $anames, resultname = $rn, links = $env)
to mcrl2::MultiAction(actions = $actions,
  result = $result)
where for $aname in $anames,
          $action in $actions :
          ($action in $env,
           $action = textualmcrl2::Atom(name =
             $aname)),
      $result in $env,
      $result = textualmcrl2::Atom(name = $rn)

from textualmcrl2::Sequence
to mcrl2::Sequence

from textualmcrl2::Implication(condition = $c)
to mcrl2::Implication
where $c.type.sort = mcrl2::BoolSort

from textualmcrl2::Number
to mcrl2::Number(type=$t)
where $t = mcrl2::Nat(sort=mcrl2::NatSort)

from textualmcrl2::BooleanLiteral
to mcrl2::BooleanLiteral(type = $t)
where $t = mcrl2::Bool(sort = mcrl2::BoolSort)

from textualmcrl2::SetEnumeration(elements =
  $es)
with $te
to mcrl2::SetEnumeration(type = $t)
where for $e in $es : $te > $e.type.sort,
      $t = mcrl2::Set
               (sort=mcrl2::SetSort
                         (elementSort=$te))

from textualmcrl2::Identifier(varname = $n,
  links = $env)
to mcrl2::Identifier(variable=$v, type = $t)
where $v in $env,
      $n = $v.name,
      $t = $v.sort

from textualmcrl2::Identifier(varname = $n,
  links = $env)
with $sort
to mcrl2::ConstructorReference(constructor=$c,
  type = $t)
where $sort in $env,
      $c in $sort.constructors,
      $n = $c.name,
      $t = mcrl2::SortRef(sort=$sort)

from textualmcrl2::ListEnumeration(elements =
  $es)
with $te
```

```
to mcrl2::ListEnumeration(type = $t)
where for $e in $es : $te > $e.type.sort,
      $t = mcrl2::List
               (sort=mcrl2::ListSort
                         (elementSort=$te))

from textualmcrl2::BagEnumeration(elements =
 $es)
with $te
to mcrl2::BagEnumeration(type = $t)
where for $e in $es :
          ($e.quantity.type.sort =
            mcrl2::NatSort,
           $te > $e.element.type.sort),
      $t = mcrl2::Bag
               (sort=mcrl2::BagSort
                         (elementSort=$te))

from textualmcrl2::UnaryExpression(right= $e,
  operation = $op)
to mcrl2::UnaryExpression(type = $t)
where $op = textualmcrl2::UnaryOps::Negation,
      $e.type.sort = mcrl2::BoolSort,
      $t = mcrl2::Bool(sort=mcrl2::BoolSort)

from textualmcrl2::UnaryExpression(right= $e,
  operation = $op)
to mcrl2::UnaryExpression(type = $t)
where $op = textualmcrl2::UnaryOps::Minus,
      $e.type.sort < mcrl2::IntSort,
      $t = mcrl2::Int(sort=mcrl2::IntSort)

from textualmcrl2::UnaryExpression(right= $e,
  operation = $op)
to mcrl2::UnaryExpression(type = $t)
where $op = mcrl2::UnaryOps::Minus,
      $e.type.sort = mcrl2::RealSort,
      $t = mcrl2::Real(sort=mcrl2::RealSort)

from textualmcrl2::UnaryExpression(right= $e,
  operation = $op)
to mcrl2::UnaryExpression(type = $t)
where $op = textualmcrl2::UnaryOps::ListSize,
      $e.type = mcrl2::ListSort,
      $t = mcrl2::Nat(sort=mcrl2::NatSort)

```

```
from textualmcrl2::Quantification(expression =
  $e)
to mcrl2::Quantification(type = $t)
where $e.type.sort = mcrl2::BoolSort,
      $t = mcrl2::Bool(sort=mcrl2::BoolSort)

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $te
to mcrl2::BinaryExpression(type = $t)
where $op =
  textualmcrl2::BinaryOps::Projection,
      $l.type.sort = mcrl2::ListSort
                         (elementSort = $te),
      $r.type.sort = mcrl2::NatSort,
      $t = mcrl2::SortRef(sort=$te)

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $t
to mcrl2::BinaryExpression(type = $et)
where $op in set{
  textualmcrl2::BinaryOps::Multiplication,
                 textualmcrl2::BinaryOps::Addition,
                 textualmcrl2::BinaryOps::Subtraction},
      $t > $l.type.sort,
      $t > $r.type.sort,
      mcrl2::RealSort > $t,
      $et = mcrl2::Real(sort=$t)

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $te,$tle,$tre
to mcrl2::BinaryExpression(type = $t)
where $op in set{
  textualmcrl2::BinaryOps::Multiplication,
                 textualmcrl2::BinaryOps::Addition,
                 textualmcrl2::BinaryOps::Subtraction},
      $l.type.sort =
        mcrl2::SetSort(elementSort = $tle),
      $r.type.sort =
        mcrl2::SetSort(elementSort = $tre),
      $te > $tle,
      $te > $tre,
      $t =
        mcrl2::Set(sort=mcrl2::SetSort(elementSort
        = $te))

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $ttemp
to mcrl2::BinaryExpression(type = $t)
where $op in set{
  textualmcrl2::BinaryOps::Modulo,
                 textualmcrl2::BinaryOps::Division},
      $ttemp > $l.type.sort,
      $ttemp > $r.type.sort,
      mcrl2::IntSort > $ttemp,
      mcrl2::Int(sort = $ttemp) = $t

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
to mcrl2::BinaryExpression(type = $t)
where $op in set{
  textualmcrl2::BinaryOps::LessThan,
                 textualmcrl2::BinaryOps::GreaterThan,
                 textualmcrl2::BinaryOps::LessEqual,
                 textualmcrl2::BinaryOps::GreaterEqual},
      mcrl2::RealSort > $l.type.sort,
      mcrl2::RealSort > $r.type.sort,
      $t = mcrl2::Bool(sort=mcrl2::BoolSort)

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $te, $tle, $tre
to mcrl2::BinaryExpression(type = $t)
where $op in set{
  textualmcrl2::BinaryOps::LessThan,
                 textualmcrl2::BinaryOps::GreaterThan,
                 textualmcrl2::BinaryOps::LessEqual,
                 textualmcrl2::BinaryOps::GreaterEqual},
      $l.type.sort = mcrl2::SetSort
                         (elementSort = $tle),
      $r.type.sort = mcrl2::SetSort
                         (elementSort = $tre),
      $te > $tle,
      $te > $tre,
      $t = mcrl2::Bool(sort=mcrl2::BoolSort)

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $te
```

```
to mcrl2::BinaryExpression(type = $t)
where $op = textualmcrl2::BinaryOps::Element,
      $l.type.sort = mcrl2::SetSort
                         (elementSort = $te),
      $te > $r.type.sort,
      $t = mcrl2::Bool(sort=mcrl2::BoolSort)

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $te
to mcrl2::BinaryExpression(type = $t)
where $op = textualmcrl2::BinaryOps::Element,
      $l.type.sort = mcrl2::ListSort
                         (elementSort = $te),
      $te > $r.type.sort,
      $t = mcrl2::Bool(sort=mcrl2::BoolSort)

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $te
to mcrl2::BinaryExpression(type = $t)
where $op = textualmcrl2::BinaryOps::Element,
      $l.type.sort = mcrl2::BagSort
                         (elementSort = $te),
      $te > $r.type.sort,
      $t = mcrl2::Bool(sort=mcrl2::BoolSort)

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $te, $tre
to mcrl2::BinaryExpression(type = $t)
where $op = textualmcrl2::BinaryOps::Cons,
      $r.type.sort = mcrl2::ListSort
                         (elementSort = $tre),
      $te > $tre,
      $te > $l.type.sort,
      $t = mcrl2::List
               (sort=mcrl2::ListSort
                         (elementSort = $te))

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $te, $tle
to mcrl2::BinaryExpression(type = $t)
where $op = textualmcrl2::BinaryOps::Snoc,
      $l.type.sort = mcrl2::ListSort
```

```
                         (elementSort = $tle),
      $te > $tle,
      $te > $r.type.sort,
      $t = mcrl2::List
               (sort=mcrl2::ListSort
                         (elementSort = $te))

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $te,$tle,$tre
to mcrl2::BinaryExpression(type = $t)
where $op in
  set{textualmcrl2::BinaryOps::Concat},
      $l.type.sort = mcrl2::ListSort
                         (elementSort = $tle),
      $r.type.sort = mcrl2::ListSort
                         (elementSort = $tre),
      $te > $tle,
      $te > $tre,
      $t = mcrl2::List
               (sort=mcrl2::ListSort
                         (elementSort = $te))

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
with $te
to mcrl2::BinaryExpression(type = $t)
where $op in set{
  textualmcrl2::BinaryOps::Equal,
                textualmcrl2::BinaryOps::DisEqual},
      $l.type.sort < $te,
      $r.type.sort < $te,
      $t = mcrl2::Bool(sort=mcrl2::BoolSort)

from textualmcrl2::BinaryExpression(right= $r,
  left = $l, operator = $op)
to mcrl2::BinaryExpression(type = $t)
where $op in set{
  textualmcrl2::BinaryOps::Conjunction,
                 textualmcrl2::BinaryOps::Disjunction,
                 textualmcrl2::BinaryOps::Implication},
      $l.type.sort = mcrl2::BoolSort,
      $r.type.sort = mcrl2::BoolSort,
      $t = mcrl2::Bool(sort=mcrl2::BoolSort)

from textualmcrl2::FunctionApplication(base =
  $b, arguments = $args)
with $ft, $params
to mcrl2::FunctionApplication(type = $t)
where $ft = $b.type,
      $ft = mcrl2::HigherOrderSort
                 (domain = $params, result =
                   $t),
      for $arg in $args,
          $param in $params :
          $arg.type.sort < $param.sort

from textualmcrl2::FunctionApplication(base =
  $b, arguments = $args)
with $ft, $params
to mcrl2::FunctionApplication(type = $t)
where $ft = $b.constructor,
      $ft = textualmcrl2::Constructor
                (projections = $params),
      for $arg in $args,
          $param in $params :
          $arg.type.sort < $param.domain.sort,
      $t = $b.type

from textualmcrl2::MapAccess(base = $b, key =
  $k, value = $v)
with $parameterlist, $parameter, $valuetype
to mcrl2::MapAccess(type=$t)
where $t = $b.type,
      $b.type.sort = mcrl2::HigherOrderSort
                          (domain =
                            $parameterlist,
                           result =
                             $valuetype),
      length $parameterlist = 1,
      $parameter in $parameterlist,
      $k.type.sort < $parameter,
      $v.type.sort < $valuetype

from textualmcrl2::Lambda(variable = $v,
  expression = $e)
with $d, $r, $vt
to mcrl2::Lambda(type = $t)
where $vt = $v.sort.sort,
      $d = set{$vt},
      $r = $e.type.sort,
      $t = mcrl2::HigherOrder
```

```
                  (sort=mcrl2::HigherOrderSort
                            (domain = $d,
                             result = $r))

   from textualmcrl2::Equation(left = $l, right=$r)
   with $t
   to mcrl2::Equation
   where $t > $l.type.sort,
         $t > $r.type.sort

   from textualmcrl2::Sequence
   to mcrl2::Sequence

   from textualmcrl2::Choice
   to mcrl2::Choice

   from textualmcrl2::Summation
   to mcrl2::Summation

   from textualmcrl2::Deadlock
   to mcrl2::Deadlock

   from textualmcrl2::Parallelism
   to mcrl2::Parallelism

   from textualmcrl2::Communication
   to mcrl2::Communication

strategy
   mcrl2::Pos < mcrl2::Nat
   mcrl2::Nat <  mcrl2::Int
   mcrl2::Int  < mcrl2::Real
   mcrl2::PosSort < mcrl2::NatSort
   mcrl2::NatSort <  mcrl2::IntSort
   mcrl2::IntSort  < mcrl2::RealSort
   mcrl2::ListSort(elementSort = $el) < mcrl2::ListSort(elementSort = $er) if $el < $er
   mcrl2::SetSort(elementSort = $el) < mcrl2::SetSort(elementSort = $er) if $el < $er
   mcrl2::BagSort(elementSort = $el) < mcrl2::BagSort(elementSort = $er) if $el < $er
strategytarget type
```

Appendix E

Type system for POOSL

Listing E.1: POOSL EMF-TL Rules

```
imports
    http://poosl.esi.nl/untypedpoosl/1.2.0;
    http://poosl.esi.nl/poosl/1.2.0;
start untypedpoosl::Poosl
typesystem
widening
    poosl::DataClass(name = $n1) <
      poosl::DataClass(name = $n2) if $n1 =
      "Integer", $n2 = "Real"
rules

from untypedpoosl::Declaration(typename = $n,
  links = $l)
to poosl::Declaration(type = $t)
where $t in $l,
      $t = untypedpoosl::DataClass(name = $n)

from untypedpoosl::NamedDataMethod(returntypename
  = $n, links = $l)
to poosl::NamedDataMethod(returnType = $t)
where $t in $l,
      $t = untypedpoosl::DataClass(name = $n)

from untypedpoosl::DataClass(extendsname = $n,
  instanceVariable = $vs, method = $ms)
to poosl::DataClass(allInstanceVariables = $vs,
  allMethods = $ms)
where $n = OclUndefined

```

```
from untypedpoosl::DataClass(extendsname = $n,
  links = $l, instanceVariable = $vs, method =
  $ms)
to poosl::DataClass(extends = $c,
  allInstanceVariables = $avs, allMethods = $ams)
where $c in $l,
      $c = untypedpoosl::DataClass(name = $n),
      $avs = $vs & ($c.allInstanceVariables),
      $ams = $ms & ($c.allMethods)

from untypedpoosl::ProcessClass(extendsname = $n,
  variable = $vs, method = $ms)
to poosl::ProcessClass(allVariables = $vs,
  allMethods = $ms)
where $n = OclUndefined

from untypedpoosl::ProcessClass(extendsname = $n,
  links = $l, variable = $vs, method = $ms)
to poosl::ProcessClass(extends = $c, allVariables
  = $avs, allMethods = $ams)
where $c in $l,
      $c = untypedpoosl::ProcessClass(name = $n),
      $avs = $vs & ($c.allVariables),
      $ams = $ms & ($c.allMethods)


from untypedpoosl::Channel(links = $l, portname =
  $pn)
to poosl::Channel(port = $p)
where $p in $l,
      $p = untypedpoosl::Port(name = $pn)

from untypedpoosl::ConnectionPI(links = $l,
  channelname = $cn, portname = $pn, instancename
  = $in)
with $class, $ports
to poosl::ConnectionPI(channel = $c, port = $p,
  instance = $i)
where $c in $l,
      $c = untypedpoosl::Channel(name = $cn),
      $i in $l,
      $i = untypedpoosl::Instance(name = $in),
      $class = $i.classDefinition,
      $class = untypedpoosl::ProcessClass(port =
        $ports),
      $p in $ports,
      $p = untypedpoosl::Port(name = $pn)


from untypedpoosl::ConnectionEP(links = $l,
  channelname = $cn, portname = $pn)
to poosl::ConnectionEP(channel = $c, port = $p)
where $c in $l,
      $c = untypedpoosl::Channel(name = $cn),
      $p in $l,
      $p = untypedpoosl::Port(name = $pn)

from untypedpoosl::Instance(links = $l, classname
  = $cn, arguments = $args)
with $params, $ptypeslist, $paramtypes, $argexps,
  $arguments
to poosl::Instance(classDefinition = $c)
where $c in $l,
      $c =
        untypedpoosl::SpecificationPrimaryClass(name
        = $cn, instantiationParameter = $params),
      for $param in $params,
          $ptypes in $ptypeslist,
          $vars  : ($param =
            untypedpoosl::Declaration(variable =
            $vars),
                   for $ptype in $ptypes,
                       $var in $vars :
                       $ptype = $param.type),
      $paramtypes = flatten $ptypeslist,
      $args =
        untypedpoosl::ListOfExpressions(expressions
        = $argexps),
      for $argexp in $argexps,
          $argument in $arguments,
          $paramtype in $paramtypes :
          ($argexp = untypedpoosl::Expressions
                         (expression = $argument),
           $paramtype > last $argument.type)


from untypedpoosl::BooleanConstant(links = $l)
to poosl::BooleanConstant(type = $t)
where $t in $l,
      $t = untypedpoosl::DataClass(name="Boolean")

from untypedpoosl::CurrentTime(links = $l)
```

```
to poosl::CurrentTime(type = $t)
where $t in $l,
      $t = untypedpoosl::DataClass(name="Real")

from untypedpoosl::IntegerConstant(links = $l)
to poosl::IntegerConstant(type = $t)
where $t in $l,
      $t = untypedpoosl::DataClass(name="Integer")

from untypedpoosl::New(links = $l, classname = $c)
to poosl::New(type = $t)
where $t in $l,
      $t = untypedpoosl::DataClass(name=$c)

from untypedpoosl::NilPrimary(links = $l)
with $c
to poosl::NilPrimary(type = $t)
where $t in $l,
      $t = untypedpoosl::DataClass(name="Nil")

from untypedpoosl::RealConstant(links = $l)
to poosl::RealConstant(type = $t)
where $t in $l,
      $t = untypedpoosl::DataClass(name="Real")

from untypedpoosl::RoundBracketExpression(expressions = $exps, links = $l)
with $exp, $e
to poosl::RoundBracketExpression(type = $t)
where $exps = untypedpoosl::Expressions(expression = $exp),
      $e = last $exp,
      $t > $e.type,
      $t in $l

from untypedpoosl::Self(links = $l)
to poosl::Self(type = $t)
where $t in $l,
      $t = untypedpoosl::DataClass

from untypedpoosl::StringConstant(links = $l)
to poosl::StringConstant(type = $t)
where $t in $l,
      $t = untypedpoosl::DataClass(name="String")
```

```
from untypedpoosl::VariablePrimary(variablename = $n, links = $l) in $dc : untypedpoosl::DataClass
with $d, $lv
to poosl::VariablePrimary(type = $t, variable = $v)
where $d in $l,
      $d = untypedpoosl::Declaration(variable = $lv),
      $d in $dc.allInstanceVariables,
      $v in $lv,
      $v = untypedpoosl::Variable(name = $n),
      $t = $d.type

from untypedpoosl::PrimaryExpression(primary=$p, methodCall = $mc, minusSign = $minus)
with $s
to poosl::PrimaryExpression(type = $t)
where $minus = "false",
      $s = length $mc,
      $s = 0,
      $t = $p.type

from untypedpoosl::PrimaryExpression(primary = $p, methodCall = $mc, minusSign = $minus) in $dc : untypedpoosl::DataClass
with $dcmethods, $firstmethod, $lastmethod, $mcpairs
to poosl::PrimaryExpression(type = $t)
where $minus = "false",
      $p = OclUndefined,
      $dc = untypedpoosl::DataClass(method = $dcmethods),
      $firstmethod = first $mc.method,
      $firstmethod in $dcmethods,
      $mcpairs = pairs $mc,
      for ($mc1,$mc2) in $mcpairs:
          ($mc2.method in $mc1.method.returnType.method),
      $lastmethod = last $mc.method,
      $t = $lastmethod.returnType

from untypedpoosl::PrimaryExpression(primary = $p, methodCall = $mc, minusSign = $minus)
with $dcmethods, $firstmethod, $lastmethod, $mcpairs
to poosl::PrimaryExpression(type = $t)
```

```
where $minus = "false",
      $p.type = untypedpoosl::DataClass(method =
        $dcmethods),
      $firstmethod = first $mc.method,
      $firstmethod in $dcmethods,
      $mcpairs = pairs $mc,
      for ($mc1,$mc2) in $mcpairs:
          ($mc2.method in
            $mc1.method.returnType.method),
      $lastmethod = last $mc.method,
      $t = $lastmethod.returnType

from untypedpoosl::PrimaryExpression(primary=$p,
  methodCall = $mc, minusSign = $minus)
with $s
to poosl::PrimaryExpression(type = $t)
where $minus = "true",
      $s = length $mc,
      $s = 0,
      $t = $p.type,
      $t = untypedpoosl::DataClass(name =
        "Boolean")

from untypedpoosl::PrimaryExpression(primary =
  $p, methodCall = $mc, minusSign = $minus) in
  $dc : untypedpoosl::DataClass
with $dcmethods, $firstmethod, $lastmethod,
  $mcpairs
to poosl::PrimaryExpression(type = $t)
where $minus = "true",
      $p = OclUndefined,
      $dc = untypedpoosl::DataClass(method =
        $dcmethods),
      $firstmethod = first $mc.method,
      $firstmethod in $dcmethods,
      $mcpairs = pairs $mc,
      for ($mc1,$mc2) in $mcpairs:
          ($mc2.method in
            $mc1.method.returnType.method),
      $lastmethod = last $mc.method,
      $t = $lastmethod.returnType,
      $t = untypedpoosl::DataClass(name =
        "Boolean")

from untypedpoosl::PrimaryExpression(primary =
  $p, methodCall = $mc, minusSign = $minus)
```

```
with $dcmethods, $firstmethod, $lastmethod,
  $mcpairs
to poosl::PrimaryExpression(type = $t)
where $minus = "true",
      $p.type = untypedpoosl::DataClass(method =
        $dcmethods),
      $firstmethod = first $mc.method,
      $firstmethod in $dcmethods,
      $mcpairs = pairs $mc,
      for ($mc1,$mc2) in $mcpairs:
          ($mc2.method in
            $mc1.method.returnType.method),
      $lastmethod = last $mc.method,
      $t = $lastmethod.returnType,
      $t = untypedpoosl::DataClass(name =
        "Boolean")

from untypedpoosl::PrimaryExpression(primary=$p,
  methodCall = $mc, minusSign = $minus)
with $s
to poosl::PrimaryExpression(type = $t)
where $minus = "true",
      $s = length $mc,
      $s = 0,
      $t = $p.type,
      $t < untypedpoosl::DataClass(name = "Real")

from untypedpoosl::PrimaryExpression(primary =
  $p, methodCall = $mc, minusSign = $minus) in
  $dc : untypedpoosl::DataClass
with $dcmethods, $firstmethod, $lastmethod,
  $mcpairs
to poosl::PrimaryExpression(type = $t)
where $minus = "true",
      $p = OclUndefined,
      $dc = untypedpoosl::DataClass(method =
        $dcmethods),
      $firstmethod = first $mc.method,
      $firstmethod in $dcmethods,
      $mcpairs = pairs $mc,
      for ($mc1,$mc2) in $mcpairs:
          ($mc2.method in
            $mc1.method.returnType.method),
      $lastmethod = last $mc.method,
      $t = $lastmethod.returnType,
      $t < untypedpoosl::DataClass(name = "Real")
```

```

from untypedpoosl::PrimaryExpression(primary =
  $p, methodCall = $mc, minusSign = $minus)
with $dcmethods, $firstmethod, $lastmethod,
  $mcpairs
to poosl::PrimaryExpression(type = $t)
where $minus = "true",
      $p.type = untypedpoosl::DataClass(method =
        $dcmethods),
      $firstmethod = first $mc.method,
      $firstmethod in $dcmethods,
      $mcpairs = pairs $mc,
      for ($mc1,$mc2) in $mcpairs:
          ($mc2.method in
            $mc1.method.returnType.method),
      $lastmethod = last $mc.method,
      $t = $lastmethod.returnType,
      $t < untypedpoosl::DataClass(name = "Real")

from untypedpoosl::DataMethodCall(links = $l,
  methodname = $mn, arguments = $args)
with $params, $ptypeslist, $varlists,
  $paramtypes, $argexps, $arguments
to poosl::DataMethodCall(method = $method)
where $method in $l,
      $method = untypedpoosl::NamedDataMethod
                    (parameter = $params, name =
                      $mn),
      for $param in $params,
          $ptypes in $ptypeslist,
          $vars: ($param =
            untypedpoosl::Declaration
                               (variable = $vars),
                  for $ptype in $ptypes,
                      $var in $vars :
                      $ptype = $param.type),
      $paramtypes = flatten $ptypeslist,
      $args = untypedpoosl::ListOfExpressions
                  (expressions = $argexps),
      for $argexp in $argexps,
          $argument in $arguments,
          $paramtype in $paramtypes :
          ($argexp = untypedpoosl::Expressions
                         (expression = $argument),
           $paramtype > last $argument.type)

```

```
error untypedpoosl::PrimaryExpression
message "Undefined Expression"

from untypedpoosl::OperatorExpression(operator =
  $o, leftOperand = $l, rightOperand = $r,links =
  $links)
with $te
to poosl::OperatorExpression(type = $t)
where $o in set{poosl::Operator::equals,
                poosl::Operator::equalsNot,
                poosl::Operator::isIdenticalWith,
                poosl::Operator::isNotIdenticalWith},
      $te = $l.type,
      $te = $r.type,
      $t in $links,
      $t = untypedpoosl::DataClass(name="Boolean")

from untypedpoosl::OperatorExpression(operator =
  $o, leftOperand = $l, rightOperand = $r)
to poosl::OperatorExpression(type = $t)
where $o in set{poosl::Operator::and,
                poosl::Operator::or},
      $t = $l.type,
      $t = $r.type,
      $t = untypedpoosl::DataClass(name =
        "Boolean")

from untypedpoosl::OperatorExpression(operator =
  $o, leftOperand = $l, rightOperand = $r)
to poosl::OperatorExpression(type = $t)
where $o in set{poosl::Operator::add,
                poosl::Operator::subtract,
                poosl::Operator::multiply,
                poosl::Operator::divide},
      $t > $l.type,
      $t > $r.type,
      $t < untypedpoosl::DataClass(name = "Real")

from untypedpoosl::OperatorExpression(operator =
  $o, leftOperand = $l, rightOperand = $r)
with $te
to poosl::OperatorExpression(type = $t)
where $o in set{poosl::Operator::lessThan,
                poosl::Operator::lessOrEqual,
                poosl::Operator::moreThan,
                poosl::Operator::moreOrEqual},
      $t > $l.type,
      $t > $r.type,
      $t < untypedpoosl::DataClass(name = "Real"),
      $t = untypedpoosl::DataClass(name = "Boolean")

from untypedpoosl::AssignmentExpression(name = $n, expression = $e, links = $l)
with $d, $dvs
to poosl::AssignmentExpression(type = $t, variable = $v)
where $d in $l,
      $d = untypedpoosl::Declaration(variable = $dvs),
      $v in $dvs,
      $v = untypedpoosl::Variable(name = $n),
      $t = $d.type,
      $t > $e.type

from untypedpoosl::IfExpression(condition = $cexps, thenClause = $texps, elseClause = $eexps)
with $ccs, $tcs, $ecs, $bool
to poosl::IfExpression(type = $t)
where $cexps = untypedpoosl::Expressions(expression = $ccs),
      $bool = last $ccs.type,
      $bool = untypedpoosl::DataClass(name = "Boolean"),
      $texps = untypedpoosl::Expressions(expression = $tcs),
      $t > last $tcs.type,
      $eexps = untypedpoosl::Expressions(expression = $ecs),
      $t > last $ecs.type

error untypedpoosl::IfExpression
message "Incorrect if expression"

from untypedpoosl::ReturnExpression(expression = $e)
to poosl::ReturnExpression(type = $t)
where $t = $e.type

```

```
from untypedpoosl::WhileExpression(condition =
  $cexps, loopBody = $lexps)
with $ccs, $lcs, $bool
to poosl::WhileExpression(type = $t)
where $cexps =
  untypedpoosl::Expressions(expression = $ccs),
      $bool = last $ccs.type,
      $bool = untypedpoosl::DataClass(name =
        "Boolean"),
      $lexps =
        untypedpoosl::Expressions(expression =
        $lcs),
      $t > last $lcs.type

from
  untypedpoosl::MessageReceiveSignature(portname
  = $n, links = $l)
to poosl::MessageReceiveSignature(port = $p)
where $p in $l,
      $p = untypedpoosl::Port(name = $n)

from untypedpoosl::MessageSendSignature(portname
  = $n, links = $l)
to poosl::MessageSendSignature(port = $p)
where $p in $l,
      $p = untypedpoosl::Port(name = $n)

from untypedpoosl::MessageParameter(parametername
  = $n, links = $l)
to poosl::MessageParameter(parameter = $p)
where $p in $l,
      $p = untypedpoosl::DataClass(name = $n)

from untypedpoosl::PortName(portname = $n, links
  = $l)
to poosl::PortName(port = $p)
where $p in $l,
      $p = untypedpoosl::Port(name = $n)

from untypedpoosl::Expressions
to poosl::Expressions

from untypedpoosl::DelayStatement(expression =
  $cexp)
with $real
to poosl::DelayStatement
```

```
where $cexp.type < untypedpoosl::DataClass(name =
  "Real")

from
  untypedpoosl::GuardedStatement(guardExpressions
  = $cexps)
with $ccs, $bool
to poosl::GuardedStatement
where $cexps =
  untypedpoosl::Expressions(expression = $ccs),
      last $ccs.type =
        untypedpoosl::DataClass(name = "Boolean")

from
  untypedpoosl::IfStatement(conditionExpressions
  = $cexps)
with $ccs, $bool
to poosl::IfStatement
where $cexps =
  untypedpoosl::Expressions(expression = $ccs),
      last $ccs.type =
        untypedpoosl::DataClass(name = "Boolean")

from untypedpoosl::ReceiveMessage(messagename =
  $n, links = $l, receptionCondition = $ce,
  variablenames = $vns)
with $msig, $bool, $ps
to poosl::ReceiveMessage(_message = $m, variable
  = $vs)
where $msig in $l,
      $msig =
        untypedpoosl::MessageReceiveSignature
                 (_message = $m, parameter = $ps),
      $m = untypedpoosl::MessageReceive(name =
        $n),
      $ce = OclUndefined,
      for $v in $vs,
          $vn in $vns :
          ($v in $l,
           $v = untypedpoosl::Variable(name =
             $vn)),
      for $v in $vs,
          $dec, $decvars,
          $p in $ps :
          ($dec in $l,
               $dec = untypedpoosl::Declaration
```

```
                         (variable = $decvars),
               $v in $decvars,
               $dec.type = $p.parameter)

from untypedpoosl::ReceiveMessage(messagename =
  $n, links = $l, receptionCondition = $ce,
  variablenames = $vns)
with $msig, $bool, $ps, $decs, $decvarslist,
  $dectypes
to poosl::ReceiveMessage(_message = $m, variable
  = $vs)
where $msig in $l,
      $msig =
        untypedpoosl::MessageReceiveSignature
                  (_message = $m, parameter =
                    $ps),
      $m = untypedpoosl::MessageReceive(name =
        $n),
      $bool = $ce.type,
      $bool =
        untypedpoosl::DataClass(name="Boolean"),
      for $v in $vs,
          $vn in $vns :
          ($v in $l, $v = untypedpoosl::Variable
                              (name = $vn)),
      for $v in $vs,
          $dec in $decs,
          $decvars in $decvarslist,
          $p in $ps :
          ($dec in $l,
           $dec = untypedpoosl::Declaration
                      (variable = $decvars),
           $v in $decvars,
           $dec.type = $p.parameter)

from untypedpoosl::SendMessage(messagename = $n,
  links = $l, listOfExpressions = $liste)
with $msig, $ms, $exps, $es, $ts, $ps
to poosl::SendMessage(_message = $m)
where $msig in $l,
      $msig = untypedpoosl::MessageSendSignature
                  (_message = $m, parameter =
                    $ps),
      $m = untypedpoosl::MessageSend(name = $n),
      for $t in $ts,
          $p in $ps :
```

```
          $t = $p.parameter,
      $liste = untypedpoosl::ListOfExpressions
                   (expressions = $exps),
      for $exp in $exps,
          $e in $es,
          $t in $ts :
          ($exp =
            untypedpoosl::Expressions(expression
            = $e),
           $t = last $e.type)

from untypedpoosl::ProcessMethodCall(links = $l,
  methodname = $mn, inputArguments = $args,
  outputvarnames = $ovnames)
with $params, $ptypeslist, $paramtypes,
     $outparams, $outptypeslist, $outparamtypes,
     $varlists,
     $argexps, $arguments, $ovdecs
to poosl::ProcessMethodCall(method = $method,
  outputVariables = $ovs)
where $method in $l,
      $method = untypedpoosl::ProcessMethod(
          inputParameter = $params,
          name = $mn,
          outputParameter = $outparams),
      for $param in $params,
          $paramvars,
          $ptypes in $ptypeslist: (
              $param = untypedpoosl::Declaration
                  (variable = $paramvars),
              for $ptype in $ptypes,
                  $var in $paramvars :
                      $ptype = $param.type),
      $paramtypes = flatten $ptypeslist,
      for $outparam in $outparams,
          $outparamvars,
          $outptypes in $outptypeslist: (
               $outparam =
                 untypedpoosl::Declaration
                   (variable = $outparamvars),
               for $outptype in $outptypes,
                  $var in $outparamvars :
                      $outptype = $outparam.type),
      $outparamtypes = flatten $outptypeslist,
      $args = untypedpoosl::ListOfExpressions
                  (expressions = $argexps),
      for $argexp in $argexps,
          $argument in $arguments,
          $paramtype in $paramtypes :
              ($argexp = untypedpoosl::Expressions
                             (expression =
                               $argument),
               $paramtype > last $argument.type),
      for $ov in $ovs,
          $ovdec,
          $ovname in $ovnames,
          $outparamtype in $outparamtypes :
              ($ovdec in $l,
               $ov in $ovdec.variable,
               $outparamtype = $ovdec.type,
               $ov = untypedpoosl::Variable
                         (name = $ovname))


from untypedpoosl::SkipStatement
to poosl::SkipStatement

from
  untypedpoosl::WhileStatement(conditionExpressions
  = $cexps)
with $ccs, $cc, $bool
to poosl::WhileStatement
where $cexps =
  untypedpoosl::Expressions(expression = $ccs),
      $cc = last $ccs,
      $bool = $cc.type,
      $bool = untypedpoosl::DataClass(name =
        "Boolean")

strategy
    poosl::DataClass(name = $n1) <
      poosl::DataClass(name = $n2) if $n1 =
      "Integer", $n2 = "Real"
strategytarget type
```

Appendix F

Summary

In recent years, Model Driven Engineering (MDE) has been suggested as an approach to make software development easier, faster and cheaper. In order to achieve this, domain models are used to represent systems and their behavior within a specific knowledge domain such as finance or complex manufacturing systems. Domain experts design and interact with these models. As domain experts frequently have limited knowledge of software engineering, facilitating model design and interaction becomes imperative. A popular solution addressing this challenge consists in creating Domain-Specific Languages (DSLs). As opposed to General Purpose Languages (GPLs), DSLs include special constructs for domain concepts, and exclude constructs that are not useful in the domain. Using these constructs, models can be made smaller, clearer and more expressive.

One disadvantage of DSLs is that each DSL needs its own set of tools to implement it in order to make the DSL effective. Tools are used to create, display and modify models, to convert models between different representations and languages, and to express the behavior or computation expressed in the model. From a theoretical perspective, these tools are all based on two fundamental aspects of any formal language, its *syntax* and *semantics*. The syntax describes how models in the language are constructed. Usually, the syntax is split into two parts, one that defines the concepts at a conceptual level, called *abstract* syntax, and a part that describes how models are presented to the user, called *concrete* syntax. Most formal languages allow a user to create models in concrete syntax, which is then converted to a representation in abstract syntax by a so-called *parser*.

A common step after parsing is *type checking*. The purpose of type checking is twofold. First of all, type checking uses the semantics of the language to recover more explicit information from the result of parsing. This more explicit information might be required for the subsequent model processing such as conversion between different representations and languages. For instance, the type information can also be used to select between different possible interpretations of similar constructs, allowing more concise representations of detailed models.

Second, type checking provides early error diagnostics. Indeed, as the name suggests, type checking revolves around the concept of types. Types describe set of values that can be produced as results of computations, or are expected by computations as input. By comparing the type of the values produced to the expected type of the input, inconsistencies can be discovered before they cause failures or errors.

These features make type systems useful for many languages. However, constructing a type checker takes significant effort and requires specific skills. Additionally, it is difficult to verify whether a manually constructed type checker indeed implements the desired type system. We propose to address this by generating type checkers based on specifications. We believe this process will reduce the skill required to create a type checker and clarify the relation between type checker and type system, making it easier both to create a type checker and to evaluate the usefulness of type system constructs for different languages. However, in order to specify type systems, a suitable formalism is needed. This formalism should be precise, but accessible to language designers, and support the language design process. This leads to the primary research question of this thesis: *How can DSL type systems be specified in an understandable, formal and evolvable way?*

To decide what properties a type system formalism needs, we conducted a Systematic Literature Review (SLR) to discover the kind of type systems DSLs have. We found DSL type systems are generally strong, static and not object-oriented. Based on the SLR, exploration of the requirements of DSL stakeholders and previous experience, we decided to select Modular Structural Operational Semantics (MSOS) as our first candidate as type system specification formalism. MSOS is a semantics formalism developed to specify operational semantics in a syntax-oriented and inductive way. Until now, MSOS has mainly been used to specify dynamic semantics, but theoretically it can be used for static semantics as well. We tested this by specifying part of the type system of the modeling language Compositional Interchange Format (CIF) in MSOS. We found that was an effective specification method, that in particular allowed the domain experts to give rapid feedback on how the type system should behave.

Once we had a type system, we wanted to create an implementation of it that could type CIF models. While doing this, we discovered that MSOS was not as convenient for specifying type systems as we initially presumed. In particular, we found MSOS had little support to define some of the properties of the CIF type system in an efficient manner, and the abstract nature of the rules make it hard to connect them to the actual data structures used by the parser and other tools, limiting interoperability.

To solve these problems, we decided to create our own language inspired by MSOS, EMF-TL. The primary theoretical difference between EMF-TL and MSOS lies in the addition of constraint conditions, which allow us to define CIF constructs that require multiple MSOS rules to define in one EMF-TL rule instead. Furthermore, EMF-TL is based on the Eclipse Modeling Framework, EMF, a framework designed to improve interaction between modeling tools by providing a common infrastructure. EMF aims to make DSL design easier by providing a “One Stop Shop" with tools that cover a variety of use cases. By basing our language on EMF, we can use the facilities the framework provides to connect with the infrastructure of the base language of our type system, CIF in our first example, directly.

We then implemented EMF-TL using the ATL transformation language and the ECL^{i}PS^e constraint engine. By using these pre-existing technologies we were able to create a flexible and reasonably efficient prototype with only limited effort. Because the CIF language is also EMF-based, we could use this implementation to create a prototype type checker for CIF models. Using this prototype, we could demonstrate that EMF-TL indeed solves the problems we had with MSOS and can effectively implement the CIF type system.

To validate EMF-TL further, we then carried out a number of other case studies, to evaluate the general applicability of EMF-TL. The three languages we choose for these case studies are WebDSL, an object-oriented web design language, mCRL2, a process language with algebraic data types, and POOSL, a process modeling language with object features. With these three languages, we covered both a representative set of different type system features, but also different levels of complexity and of existing type checkers. We implemented partial or complete type systems for all three languages, demonstrating the flexibility of EMF-TL.

We conclude that EMF-TL reinforces EMF as a “One Stop Shop" solution for defining DSLs, enabling the definition of non-trivial DSL type systems that can be seamlessly integrated with existing Ecore infrastructure. Overall, we have demonstrated that type system specifications are an effective and practical technique in domain specific language design.

Appendix G

Samenvatting

In recente jaren is Model-gedreven Ontwikkeling opgekomen als een gesuggereerde aanpak om het ontwikkelen van software makkelijker, sneller en goedkoper te maken. Om dit te bereiken worden domein modellen gebruikt om systemen en hun gedrag te beschrijven binnen een specifiek domein, zoals geldzaken of assemblage van complexe machinerie. Experts uit het domein ontwerpen en werken actief mee aan deze modellen. Omdat domeinexperts vaak beperkte kennis hebben van de constructie van software, is het van fundamenteel belang het ontwerpen en werken met deze modellen te faciliteren. Een populaire oplossing om deze uitdagingen aan te pakken bestaat uit het creëren van domein-specifieke talen(DST), ook wel Domain-Specific Languages (DSLs) genoemd. In tegenstelling tot algemene programmeertalen, ook wel General Purpose Languages (GPLs), bevatten domein-specifieke talen speciale constructies voor concepten in het domein, en worden constructies die niet zinvol zijn in het domein weggelaten. Met deze constructies kunnen modellen kleiner, duidelijker en expressiever gemaakt worden.

Een nadeel van DSTs is dat elke DST zijn eigen set gereedschappen nodig heeft om de DST bruikbaar te maken. Deze gereedschappen worden gebruikt om modellen te maken, te visualiseren, aan te passen, te vertalen tussen verschillende representaties en naar andere talen en om het gedrag en de berekeningen die in het model zijn uitgedrukt uit te voeren. Van een theoretisch perspectief gezien zijn al deze gereedschappen gebaseerd op twee fundamentele aspecten van elke formele taal, namelijk de *syntax* en de *semantiek.* De syntax beschrijft hoe modellen in de taal zijn opgebouwd. In veel gevallen wordt de syntax gescheiden in twee delen, een dat de concepten van de taal op een conceptueel niveau definieert, ook wel de *abstracte* syntax genaamd, en een deel dat beschrijft hoe de modellen worden getoond aan de gebruiker, ook wel *concrete* syntax genaamd. De meeste formele talen laten de gebruiker modellen maken met concrete syntax, die daarna worden vertaald naar een representatie in abstracte syntax door een zogenoemde *parser.*

Een veelgemaakte stap na parseren is *typecontrole.* Het doel van typecontrole is tweevoudig. Ten eerste, typecontrole gebruikt de semantiek van de taal om meer expliciete informatie te vergaren op basis van het resultaat van de parser. Deze extra expliciete informatie kan nodig zijn voor latere model operaties zoals vertalingen naar andere representatie en talen. Bijvoorbeeld, de type informatie kan ook gebruikt worden om te kiezen tussen verschillende mogelijke interpretaties van vergelijkbare constructies, waardoor meer compacte representaties van modellen mogelijk worden. Ten tweede, typecontrole maakt het mogelijk fouten in een vroeg stadium te detecteren. Zoals de naam suggereert draait typecontrole om het concept type. Types beschrijven sets van waardes die door berekeningen als uitkomst kunnen worden geproduceerd. Door het type van de geproduceerde waardes te vergelijken met de verwachte waarden van de invoer, kunnen inconsistenties worden ontdekt voor dat ze problemen en fouten kunnen veroorzaken.

Deze eigenschappen maken typesystemen nuttig voor veel talen. Helaas kost het construeren van een gereedschap om typecontrole uit te voeren significante moeite en vereist specifieke vaardigheden. Verder is het moeilijk om te verifiëren of een handmatig geconstrueerd typecontrolegereedschap inderdaad het gevraagde typesysteem implementeert. We stellen voor om dit aan te pakken door typecontrolegereedschappen te genereren op basis van specificaties. We geloven dat dit proces de vereiste vaardigheden om een typecontrolegereedschap te bouwen vermindert en de relatie tussen het typecontrolegereedschap en het typesysteem kan verduidelijken. Maar, om typesystemen te specificeren is een geschikt formalisme nodig. Dit formalisme moet precies zijn, maar toegankelijk voor taal ontwikkelaars, en moet het ontwikkelproces van talen ondersteunen. Dit leidt tot de primaire onderzoeksvraag van dit proefschrift: *Hoe kunnen DST typesystemen worden gespecificeerd in een begrijpelijke, formele en verbeterbare manier?*

Om te beslissen welke eigenschappen een formalisme voor typesystemen nodig heeft, hebben we een systematisch onderzoek van de literatuur, ook wel Systematic Literature Review (SLR) genoemd, uitgevoerd om te ontdekken wat voor typesystemen DSTs hebben. We ontdekten dat DST typesystemen over algemeen sterk, statische en niet object-georiënteerd zijn. Op basis van het literatuuronderzoek, bestudering van de verwachte eisen van betrokkenen bij DSTs en bestaande ervaring, besloten we Modular Structural Operational Semantics (MSOS) te selecteren als onze eerste kandidaat typesysteem specificatie formalisme. MSOS is een semantiek-formalisme ontwikkeld om operationele semantiek in een syntax-georiënteerde en inductieve manier te specificeren. Tot heden is MSOS toegepast om dynamische semantiek te beschrijven, maar het kan ook gebruikt worden voor statische semantiek. We hebben dit getoetst door delen van het typesysteem van de modeleringstaal Compositional Interchange Format (CIF) in MSOS te specificeren. We constateerden dat MSOS een effectieve specificatie-methode was, die met name de domeinexperts toestond snel te oordelen of het typesysteem voldeed aan de verwachtingen.

Toen we een typesysteem hadden, wilden we een gereedschap creëren dat gebruikt kon worden om CIF-modellen te typeren. Terwijl we dit deden, ontdekten we dat MSOS niet zo geschikt was als we eerst dachten. We realiseerden met name dat MSOS weinig ondersteuning had om sommige gewenste eigenschappen van het CIF-typesysteem in een efficiënte manier te definiëren, en dat de abstracte natuur van de regels het maken van verbindingen met bestaande data structuren die door de parser en andere gereedschappen gebruikt worden, waardoor interoperabiliteit beperkt wordt.

Om deze problemen op te lossen, besloten we onze eigen taal te creëren op basis van inspiratie uit MSOS, genaamd EMF-TL. Het voornaamste theoretische verschil tussen EMF-TL en MSOS zit in het toevoegen van beperkingscondities, die ons toestaan om CIF-constructies die meerdere MSOS regels nodig hebben in een enkele EMF-TL regel gevangen kunnen worden. Verder is EMF-TL gebaseerd op Eclipse Modeling Framework, EMF, een raamwerk ontworpen om interactie tussen modeleringstools te verbeteren door een gedeelde infrastructuur aan te bieden. EMF heeft tot doel DST ontwerp eenvoudiger te maken door een "Alles In Een" oplossing aan te bieden met tools die een variëteit aan mogelijke doeleinden afdekken. Door onze taal op EMF te baseren, kunnen we de faciliteiten gebruiken die het raamwerk aanlevert om een directe verbinding te maken met de infrastructuur van de onderliggende taal van ons typesysteem, CIF in ons eerste voorbeeld.

We hebben daarna EMF-TL geïmplementeerd door gebruik te maken van de ATL transformatietaal en het ECL^{i}PS^e systeem. Door deze bestaande technologie te gebruiken waren we in staat om een flexibel en redelijk efficiënt prototype te maken met een relatief beperkte investering. Omdat de taal CIF ook EMF gebaseerd is, konden we deze implementatie gebruiken om een prototype typecontrolegereedschap te maken voor CIF-modellen. Gebruik makend van dit prototype konden we aantonen dat EMF-TL de problemen oplost die we hadden met MSOS en het typesysteem van CIF effectief kan implementeren.

Om EMF-TL verder te valideren, hebben we daarna een aantal praktische studies uitgevoerd om de meer generieke toepasbaarheid van EMF-TL te toetsen. De drie talen die we uitkozen voor deze praktische studies zijn WebDSL, een object-georiënteerde taal voor webontwerp, mCRL2, een proces taal met algebraïsche datatypes, en POOSL, een procesmodeleringstaal met bepaalde object constructies. Met deze drie talen dekten we een representatieve set van verschillende typesysteemeigenschappen, maar ook verschillende niveaus van complexiteit en van bestaande typecontrolegereedschappen. We implementeerden gedeeltelijke of complete typesystemen voor alle drie talen, en demonstreerden hiermee de flexibiliteit van EMF-TL.

We concluderen dat EMF-TL EMF als "Alles In Een" oplossing voor DST ontwikkeling versterkt, door het mogelijk te maken niet-triviale DST typesystemen te definiëren die naadloos geïntegreerd kunnen worden met bestaande Ecore infrastructuur. In vogelvlucht hebben we gedemonstreerd dat typesysteemspecificaties een effectieve en praktische techniek is in het ontwikkelen van een domein-specifieke taal.

Curriculum Vitae

Arjan van der Meer was born in Voorburg on September 2nd 1984. After completing his secondary education at St. Oelbert grammar school, he studied computer science at the Eindhoven University of Technology. He received a bachelor degree in 2006 and a master degree in 2008. During this period, he also worked as a student assistent for the university. In 2008, he started a PhD project at the Department of Mathematics and Computer Science of the same university. The results of the project are presented in this thesis. He is currently working for the software company Nspyre as a software engineer.

Titles in the IPA Dissertation Series since 2008

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of

Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata

Theory and Modal Logic. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*.

Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*.

Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial*

Control Systems. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10