THESIS ON INFORMATICS AND SYSTEM ENGINEERING C57

Higher-Order Attribute Semantics of Flat Languages

PAVEL GRIGORENKO

TUT PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Institute of Cybernetics

This dissertation was accepted for the defence of the degree of Doctor of Philosophy in Engineering on September 28, 2010

Supervisor: Enn Tõugu, D.Sc
Institute of Cybernetics
Tallinn University of Technology
Tallinn, Estonia

Opponents: Prof. Mihhail Matskin
School of Information and Communication Technology (ICT)
Royal Institute of Technology (KTH)
Stockholm, Sweden

Prof. Merik Meriste
Faculty of Science and Technology, Institute of Technology
University of Tartu
Tartu, Estonia

Defence: November 17, 2010

Declaration: Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not previously been submitted for any degree or examination.

/Pavel Grigorenko/

ISSN 1406-4731
ISBN 978-9949-23-036-5

INFORMAATIKA JA SÜSTEEMITEHNIKA C57

Lamedate keelte kõrgemat järku atribuutsemantika

PAVEL GRIGORENKO

TTÜ
KIRJASTUS

Abstract

The thesis concerns a software development method for constructing programs from models (given by textual as well as visual specifications). The advantage of the method is that once a specification is complete and a computational goal is given, the synthesis of programs can be performed automatically.

Flat languages are introduced as a class of declarative languages for specifying single concepts as well as larger systems in a domain-specific manner. Given a problem statement, a program can be obtained from such a specification that computes a required goal. Flat languages are structurally simple, however their expressive power is sufficient to support hierarchy, inheritance and also control structures.

Higher-order attribute models are defined to provide the semantics of specifications in flat languages. These attribute models can contain simple and higher-order attribute dependencies. Simple dependencies have attributes as inputs whereas higher-order dependencies also include *subtasks* that need to be solved during the attribute evaluation. On the one hand, subtasks introduce exponential complexity, on the other hand, they enable us to synthesize recursive, branching and looping programs. We present higher-order attribute evaluation planning algorithm using a concept of *maximal linear branches*. To synthesize efficient programs, an *optimization* algorithm is given. A concrete flat language is described, and its semantics is presented by means of higher-order attribute models.

The approach is implemented in the software system CoCoViLa that is intended for the development and the usage of domain-specific languages. It adds another powerful dimension to flat languages – visual specification of programs. Specifications are written in Java classes and realizations of attribute dependencies are implemented as Java methods. This enables one to specify software declaratively and also take the advantage of the full power of an imperative language.

Algorithm of attribute evaluation on higher-order attribute models can be explained also in terms of logic, and vice versa – theorems of intuitionistic propositional calculus can be encoded in the form of higher-order attribute models. Hence it is interesting to compare the algorithm implemented in CoCoViLa with well-known theorem provers. This is done in the thesis, and the results show that only two of the provers performed better than the CoCoViLa's algorithms.

Lühikokkuvõte

Tarkvara kasvav keerukus nõuab vahendeid, mis peaksid lihtsustama tarkvaraarendust. Antud väitekiri käsitleb tarkvaraarenduse meetodit ja sellel põhinevaid vahendeid, kus programmide konstrueerimisel kasutatakse tekstilisi ja visuaalseid mudeleid. Nimetatud meetodi eeliseks on, et valminud spetsifikatsiooni järgi sünteesitakse programm täisautomaatselt. Töös on defineeritud lamedate keelte klass ja on loodud nendele keeltele realiseerimise vahendid. Lamedad keeled on deklaratiivsed keeled nii valdkonnaspetsiifiliste mõistete kui ka keeruliste süsteemide spetsifitseerimiseks. Lamedate keelte struktuur on lihtne, samas on nad piisavalt väljendusrikkad hierarhia, pärimise, polümorfismi ja põhiliste juhtimisstruktuuride esitamiseks.

Töös on kirjeldatud kõrgemat järku atribuutmudelid lamedate keelte semantika esitamiseks. Mudelid sisaldavad kahte liiki – lihtsaid ja kõrgemat järku – atribuutsõltuvusi. Lihtsate sõltuvuste sisenditeks on ainult atribuudid, kuid kõrgemat järku sõltuvuste sisendites lisanduvad atribuutidele ka alamülesanded, mis atribuutide arvutamisel peavad olema lahendatud. Ühest küljest, alamülesanded nõuavad eksponentsiaalse keerukusega arvutusi semantika realiseerimisel, teisest küljest võimaldavad nad sünteesida rekursiivseid, hargnevaid ning tsüklitega programme. On esitatud kõrgemat järku atribuutide arvutamise algoritm. Sammude arvu mõttes minimaalsete programmide saamiseks on antud optimeerimisalgoritm. Töös on esitatud konkreetne lame keel, mille avaldised on lihtsalt teisendatavad atribuutmudeliteks.

Pakutud meetod on realiseeritud Java-põhises valdkonnaspetsiifiliste keelte arendamisele orienteeritud tarkvarasüsteemis CoCoViLa. Süsteem võimaldab programme spetsifitseerida visuaalselt, hõlbustades märgatavalt lamedate keelte kasutamist. Mõistete spetsifitseerimine toimub Java klassides ning atribuutsõltuvuste realisatsiooni kirjeldatakse Java meetoditega. Antud lähenemine võimaldab tarkvara spetsifitseerida deklaratiivselt, kombineerides seda imperatiivse keele võimalustega.

Kõrgemat järku atribuutmudelitel atribuutide arvutamise algoritmi saab esitada abstraktselt ka loogika keeles ning vastupidi – intuitsionistliku lausearvutusloogika teoreeme saab sõnastada kõrgemat järku atribuutmudelite terminites. Seetõttu on huvitav võrrelda antud töös realiseeritud algoritmi loogikas tuntud tõestajatega. Võrdluse tulemused näitavad, et valitud teoreemide tõestamisel oli ainult kaks tõestajat CoCoViLas realiseeritud algoritmist paremad.

Acknowledgments

I would like to express my deep and sincere gratitude to my supervisor Enn Tyugu (D.Sc, member of the Estonian Academy of Sciences). His wide knowledge, energy, optimism, encouragement and understanding have been of a great value for me. His guidance, advice and support during my studies at the Institute of Cybernetics made this thesis possible.

I wish to thank all my colleagues from the Institute of Cybernetics for making me a very enjoyable company. My deep appreciation to Prof. Tarmo Uustalu for his enormous scientific and administrative work, eagerness to help and very interesting theoretical seminars. Gratitude to Prof. Jaan Penjam for giving me an opportunity work in such pleasurable environment and making me believe already since bachelor studies that software engineering has no value without theoretical foundations of computer science. Thanks to everybody who worked with me on CoCoViLa project – Ando Saabas, Mait Harf, Andres Ojamaa, Vahur Kotkas and Riina Maigre, it was unforgettably valuable experience. I would like to thank James Chapman, Jelena Sanko and Margarita Spichakova for both fruitful scientific discussions and off-topic.

I express gratitude to Prof. Mihhail Matskin and Prof. Merik Meriste for accepting to be my opponents and thoroughly reading this dissertation.

I warmly thank Margus Veanes for inviting me for an internship to Microsoft Research. Three months spent in Redmond were highly productive and fun. Meeting and working together with world-class experts was the experience I will never forget.

Financial support from the Estonian Ministry of Education and Research (target-financed theme No. 0322709s06), Estonian Science Foundation (grant No. 6886), EITSA Tiger University programme and ERDF (Estonian Centre of Excellence in Computer Science, EXCS) is gratefully acknowledged.

I owe my loving thanks to my family, without their encouragement it would have been impossible for me to finish this work. My dear wife Olga and son Aleksandr, for their patience and understanding and bringing joy and sense to my life. My parents, for their loving support throughout my entire life. My father, Valeri Grigorenko (D.Sc, Professor), has always been a true inspiration for me. My mother, Tatjana Senkevich, always trying hard to give the best to her son. Gratitude to my sisters Inga and Ingrid for their warm encouragement.

Contents

Chapter 1

Introduction

In this dissertation we describe a class of declarative specification languages that we call *flat languages* and present a concrete implementation of a flat language. We demonstrate through our implementation and experiments that flat languages are useful in the model-based software development. Flat languages still support hierarchy and inheritance. The term "flat" comes from the fact that any specification written in a flat language can be unfolded into a flat representation.

We present a precise semantics of flat languages in terms of higher-order attribute models. A specification in a flat language is, first, unfolded and translated into attribute model, second, attribute evaluation planning is performed and, third, evaluation algorithm is executed computing the specified goals. This approach fits well into model-based software development paradigm: models are executable specifications written using a declarative language, programs from such specifications are constructed automatically.

Institute of Cybernetics has a large experience in developing program synthesis tools. *Structured synthesis of programs* (SSP) proposed by Tyugu and Mints is a deductive synthesis method developed since late 70's. The method relies on the implicative fragment of intuitionistic propositional logic – specifications are regarded as logical theorems and programs are obtained from constructive proofs. SSP is based on the idea that programs can be constructed from preprogrammed modules taking into account only structural properties of programs being synthesized. It has been implemented in several (including commercial) software systems, such as XpertPriz, Priz, NUT, etc.

SSP has proved its usability over several decades, however some problems re-

garding its implementations remained open. The major problems were as follows. Older software systems were built for fixed platforms, consequently, the software was hardly portable and also had memory limitations. Another problem was interoperability, that is, older systems were not flexible enough to easily communicate with other software systems.

By the end of 90's the Java programming language started gaining popularity. It opened the horizon for highly portable multi-platform software, the ease of software development and effective integration between different software systems. The first attempt to implement a Java-based program synthesis system that included extensions to the structural synthesis of programs was made by Sven Lämmermann from the Royal Institute of Technology (Stockholm). The implementation remained on the prototype stage, but the experience obtained was very valuable. That included the annotation of Java classes with metainterfaces.

A new research project was started at the Institute of Cybernetics in 2003 constituting a new generation software system that relied on the ideas of structural synthesis of programs. The idea was to use Java to overcome the limitations for older systems and introduce a multi-platform model-based program synthesis framework with emphasis on visual specifications. The initial author of the implementation was Ando Saabas. The author of the present thesis joined the project in the beginning of 2004. A year later, in 2005, the software prototype was named CoCoViLa (Compiler-Compiler for Visual Languages). In 2006, the Modeling and Simulation Group was formed at the Institute of Cybernetics which aimed to further develop CoCoViLa to overcome the prototype stage and make it useful for the real-world applications. The author became the principal administrator and developer of the tool, and his contributions are summarized in the implementation part of the current thesis.

The incentive of the theoretical part of the thesis is as follows. In 1968, Donald Knuth introduced attribute grammars for the precise representation of semantics of programming languages. Later, attribute grammars became widely used in compiler construction helping to attach semantic values to syntax constructions. This approach motivated the author to take an advantage of attribute models for representing semantics of declarative domain specific languages. This required an extension of the attribute models by introduction of higher-order attribute dependencies.

One of the main advantages is the algorithmic nature of methods for processing the attributes. Structural synthesis of programs, for instance, as a pure logic-based tool, was lacking a description of how to implement the derivation rules and program extraction. Attribute evaluation algorithms, on the other hand, are more intuitive to implement.

1.1 Problem statement

The thesis concentrates on the following problems:

- defining a class of declarative specification languages suitable for model-based and automated program construction, and with a potential for visual representation;
- choosing/developing a technique for representing semantics of such languages;
- designing and implementing a specification language of this class with required extensions;
- performing experiments to show the practical applicability of the chosen approach.

1.2 Contributions

The main contributions of this work are:

- introduction of a concept of flat languages;
- method of higher-order attribute semantics for representing the semantics of flat languages;
- efficient implementation of the method in a programming environment;
- testing the programming environment in numerous applications.

1.3 Outline of the thesis

This thesis is organized as follows. Chapter 2 describes the related work and presents a context of the research. In Chapter 3 we present the formal definition of *flat languages* and give several illustrative examples. Chapter 4 is dedicated to the definition of attribute models. Simple and higher-order attribute models are introduced and algorithms of attribute evaluation on attribute models are presented. In Chapter 5 we introduce the concrete instance of a flat language – the *specification language* which consists of a *core language* and some extensions. We also define the semantics of the core language in terms of attribute models. In Chapter 6 the implementation of the proposed method of program construction is presented in the context of CoCoViLa programming environment. Chapter 7 demonstrates real-world applications in CoCoViLa. The performance evaluation and comparisons to other software tools is given in Chapter 8. In Chapter 9 we present conclusions.

Chapter 2

Related work

The present chapter defines the context of the research by giving an overview of the related work. The first section discusses program synthesis methods and some developed systems. The next section explains which attribute techniques influenced our work. The third section is about the model-based software development. Finally, a brief summary of tools for implementing domain-specific (visual) languages is given.

2.1 Program synthesis

Program synthesis is the derivation of a program to meet a given specification. This section describes several approaches and tools for program synthesis. The topic of this dissertation is closely related to the deductive synthesis ("proofs-as-programs" paradigm) based on Curry-Howard isomorphism [30]. The deductive synthesis allows to construct programs with proving its correctness with respect to the given specification.

Manna and Waldinger introduced the *deductive tableau* method for synthesis of functional programs in classical first-order logic [47, 48]. Deductive tableau is a two-dimensional structure where each row contains a single assertion or a goal, and one or more (optional) output terms which represent the program being constructed. AMPHION [42] is a knowledge-based software engineering system developed at NASA which incorporates deductive tableau method for automatically constructing programs.

Structural synthesis of programs (SSP) is a deductive program synthesis method.

SSP was first introduced by Tyugu [82] and further developed in cooperation with Mints [60]. SSP is based on the idea that programs can be constructed from preprogrammed modules taking into account only structural properties of programs being synthesized. This method has been used in several (also commercial) systems, such as XpertPriz, PRIZ [62] and NUT [87]. The foundation for SSP is the implicative fragment of intuitionistic propositional calculus (IPC). For proving theorems in the framework of SSP, instead of natural deduction, special *structural synthesis rules* (SSR) are used that are admissible rules of the IPC. SSR rules are complete and any intuitionistic propositional formula can be proved by SSR (special encoding of formulas is required). Provability in IPC and, consequently, in SSP is polynomial-space complete [78], however efficient proof strategies exist that for most of the practical problems reduce the search to almost linear complexity [51]. The discussion how SSP influenced on the current work will be continued in the following chapters.

As noted in [88], SSP is related to programming in type theories [59, 63]. Programming in type theories focuses on verification of programs against specifications and transformational development of specifications into programs. The type languages used can be very expressive compared to the language of structural synthesis, the main objective of which has been efficient automatic synthesis, setting very restrictive constraints on the type language. Nuprl [4] is a system developed at Cornell University based on the Martin-Löf's type theory [49] and the research of Constable [6].

Lämmermann [40] extended structural synthesis of programs with more complex specification mechanisms required for fully automated runtime composition of service programs. First extension was the *disjunction connective* that enabled handling of exceptions, i.e. exceptions are specified as proper outputs of preprogrammed components and are handled as regular branches in the synthesized programs. Second extension was the use (with some restrictions) of *first-order quantification* for composing programs from components of different sources. Third extension was the *falsity constant* needed to provide a way to specify program termination after exception handling. The work has been realized in the context of an object-oriented programming language. *Metainterfaces* as logical specifications of classes have been introduced. The results of Lämmermann's thesis has influenced the present work,

especially the implementation part.

Further we are going to describe two synthesis systems developed at Kestrel Institute that utilize transformational synthesis (specification refinement into programs).

KIDS (Kestrel Interactive Development System) [75, 76] semi-automatic synthesis tool that enables one to transform a formal specification into a correct and efficient program by interactively applying a sequence of high-level transformations. The system emphasizes the application of complex high-level transformations that perform significant and meaningful actions. From the user's point of view the system allows the user to make high-level design decisions like, "design a divide-and-conquer algorithm for that specification" or "simplify that expression in context".

After KIDS, Specware [52, 77] is a next generation software system developed at Kestrel Institute that uses notions and procedures based on category theory and related mathematics to manipulate specifications. In Specware, abstract specifications define the input-output relations and related properties of software under construction. Dependencies between components are defined using morphisms that syntactically map symbols from one component into symbols or terms in another component with some semantic restrictions. The abstract specifications are refined into constructive forms and abstract sorts are refined into concrete data types. As a final step, refined components are converted into components in some executable programming language (e.g., C, C++, Lisp, or Java) using a code generator.

2.2 Attribute grammars

Attribute grammars have been initially introduced by Knuth in 1968 to add semantics to context-free languages [38]. Context-free grammars describe syntax of context-free languages. Attribute semantics is defined by decorating grammars with attributes and specifying semantic functions for evaluation of attributes. This technique has been widely applied in automated systems for parser generation and compiler construction [22, 28, 68].

A survey of attribute grammar definitions and systems is given by Deransart *et al.* [15] and a survey of attribute grammar-based specification languages is given by

Paakki [67]. This is classics of compiler construction and language specification.

There have been attempts to improve/extend the concept of attribute grammars. An extension called higher-order attribute grammars (HAGs) have been theoretically considered by Vogt and Swierstra [79]. They have defined a class of higher-order attribute grammars as an extension of classical attribute grammars in the sense that parts of the parse tree can be stored in an attribute, and a parse tree itself can be changed by attribute evaluation. The strict separation between attributes and parse trees is removed in HAGs. This adds considerable flexibility to the grammar. Authors show that pure HAGs have expressive power equivalent to Turing machines. The incremental attribute evaluation algorithm for HAGs is introduced that handles the higher-order case. The problem why HAGs did not become widely used regardless their expressiveness is the fact that it is hard to reason about abstract syntax trees, structure of which may change during the attribute evaluation.

The relation between attribute grammars and declarative programs has been investigated by several researchers. Deransart and Maluszynski [16] demonstrated the relation with logic programs by introducing constructions which transform logic programs into semantically equivalent attribute grammars, and vice versa. Parigot *et al.* [17] show close relation to functional programs by introducing notions of *scheme productions* and *conditional productions* to add expressiveness to attribute grammars. The result was a declarative language with the expressive power compared to the most first-order functional languages [34].

The connection between attribute grammars and computational model introduced by Tyugu [83] has been exploited by Penjam [69]. He shows how attribute semantics of programming languages can be presented by means of computational models and proves the semantic equivalence between attribute models and computational models, both being two similar approaches to program and compiler specification and implementation. The proposed approach has been implemented using the NUT system. Productions were implemented as classes and semantic functions as computational relations between variables (attributes). In the further research, Meriste and Penjam [55, 56] introduced *attributed automata* as an extension of finite automata and a formal model for specification of software systems. An attributed automaton is a state transition system with attributes and computational relations

attached to states and transitions respectively. AAs can be used for implementation of transformational and also reactive systems. In the former case one should be careful about the existence of reachable final configurations whereas for the latter case the termination is not important. Language recognizers is one of the application of attributed automata. The latest research on *interactive attributed automata* is concerned with the application to the modeling and implementation of domain-specific languages in the context of interaction-centered multi-agent systems [53, 54].

The work of Meriste and Penjam showed that it is convenient to apply the notion of attributes not only to grammars, but also to other formalisms. In the present thesis we use attributes in the context of models of computations, rather than grammars (in the last example of Section 3.5 we will briefly discuss how attribute grammars can be represented using flat languages). Another difference of our approach from attribute grammars is that attribute models presented in this thesis do not require *well-formedness* property [15] for attribute evaluation (see Section 4.3).

2.3 Model-based software development

The languages that we define are suitable, first of all, in model-based software development. Model-based software development in general can be defined as a methodology that splits the construction of domain-specific software into two phases: domain engineering and application engineering. During the first phase, the domain concepts with high-level of abstraction are built directly using expert knowledge. The second phase uses such domain concepts to build the actual software. In the presence of appropriate tools, model-based software development facilitates high-level modeling of systems, code reuse and automated code generation.

The idea of model-based software development is not new. It has been around for almost 30 years, but has not become a widely accepted paradigm. Its most successful applications are in simulation software, there are well known specialized products like Simulink [10].

One continuous effort in the field of model-based software development is pursued in NASA [7]. Aerospace applications, including software for the International Space Station (ISS), use model-based development extensively. NASA puts strict

requirements on the model-based software development. As the authors say, *the production-quality program synthesis is the keystone for full-cycle model-based programming*. Without this capability, model-based programming is limited to being a prototyping tool whose utility ends after detailed design, when the production code is developed manually. Further, any subsequent upgrade or maintenance fix must be made manually, directly on the software. In other words, the generated code should be correct and reliable, and also modular and easily maintainable on a higher level, i.e. on the level of models, not source code.

In the work by Sztipanovits et al., domain knowledge for the signal-processing system is expressed as formal declarative models [1]. The approach stems from the use of macro-dataflow computation model in the Multigraph Architecture, which provides the framework for model-based synthesis of software in real-time, parallel-computing environment. The models are defined using a frame language in a Lisp-like syntax. Models can also be built graphically, using icons, ports and connections. The techniques developed in [1] authors still use in their latest research [71].

Considerable amount of work is being done in improving the existing UML-based approaches with the aim of providing automated support to the software development [20] and language development [74]. This approach is also related to Model Driven Architecture (MDA) advocated by the OMG group [64] and to the development of domain specific languages (DSL) [57, 89], because they all have the development of user friendly and automated problem solving tools as a goal. This approach includes the usage of UML-based models and metamodels. It concentrates either on the research of transformation rules for transforming an initial specification (a model) into another model or an executable code [94], or on the development of rules that represent the operational semantics [20] or even immediately perform the required computations.

The paper by Dionisio de Niz [14] compares UML with the Architecture Analysis and Design Language (AADL) in the context of model-based software engineering of embedded systems. The paper points out the inconsistency of UML diagrams for expressing different aspects of a system as a whole (e.g. it cannot fully define the relationships between the diagrams). AADL developed in Carnegie Mellon University is a specification language (and also a SAE Standard) that allows to define the

textual representations of software architecture and also to formally define the syntax and semantics. In addition, AADL enables to represent the system graphically. Descriptions in AADL can be verified by the syntactic and semantic analyzer for consistency and correctness. The execution semantics of AADL language is defined as *hybrid automaton*, which is a mathematical model for describing in unambiguous way how software and physical processes interact.

In [35] Kelly and Tolvanen explain how model-based code generation can be achieved. The proposed technique is called Domain-Specific Modeling (DSM). It enables to specify models using elements created specifically to describe parts of a particular domain. The DSM language follows domain abstractions and semantics and allows modelers to work directly with domain concepts. The full code is generated from high-level specifications. The underlying formalism used to describe models in one of the examples (digital wristwatch) is the *state machine*.

The important topic relevant to model-based software development is a *generative programming* paradigm. It is about manufacturing software products out of components in an automated way [9]. The generative programming focuses on software system families rather than one-of-a-kind systems. Such family members can be automatically generated based on a common *generative domain model* that includes *implementation components* and the *configuration knowledge* mapping between a specification and a finished system or component. The difference between generative programming and program synthesis is that the latter focuses more on the employment of formal methods (e.g. theorem proving) whereas the former uses rewriting and transformational approach.

2.4 Domain-specific visual languages

Over the years of rapid growth of technologies and computing power lots of systems have emerged to help software developers to reduce time and effort in creating complex software systems. In order to make programs more concise, easier to understand, write and support, notions of higher levels of abstraction are required. Visual programming/specification systems are important because visual representations of problems in some particular domain enable one to use humans visual perception

(and other cognitive processes) to examine a given picture and get a general understanding of it in a matter of seconds. Another aspect is that manipulating visual objects is more natural for humans than dealing with textual notations.

Languages that are tailored for a particular domain are called *domain-specific languages* (DSL) [90]. DSLs allow domain experts to express their knowledge and specify problems using domain-specific notations and abstractions. Domain experts may not be software engineers, thus, dealing with DSLs is more convenient than using general-purpose languages to specify domain-specific problems, as such specifications contain too many details. Examples of well-known domain-specific languages are: SQL – database query language, Yacc grammars for parser generation, XSLT – language for XML transformations, etc. Most of the DSLs are *declarative*.

Domain-specific visual languages (DSVL) give both benefits to the users – declarative and visual way to specify domain problems. There are lots of underlying concepts used to give syntax and semantics of visual languages, such as grammar-based, logic-based, graph grammars, constraint based, etc.

A software development environment NUT [87], a predecessor of a tool developed in the context of this thesis, combines object-oriented programming, visual programming, and automatic program construction paradigms. It enables users to write classes in an object-oriented style and also to specify classes graphically by drawing their schemes. Using the NUT's scheme editor, a user can develop its own graphical language and use it for specifying programs in his problem domain. The (automatic) way from a graphical specification to expected results is the following: *graphical scheme* $\rightarrow$ *textual specification* $\rightarrow$ *constraint network* $\rightarrow$ *synthesized algorithm* $\rightarrow$ *results of computation.*

AToM^3 [11, 12] (A Tool for Multi-Formalism and Meta-Modeling) is an interactive multi-paradigm modeling tool that relies on graph rewriting techniques and graph grammars to perform the transformations between formalisms (defined in *Formalism Transformation Graph* proposed by the authors) as well as for other tasks, such as code generation and operational semantics specification. AToM^3 has a meta-modeling layer in which different formalisms are modeled graphically. AToM^3 uses the Object Constraint Language OCL [93] also used in the UML to express constraints in a textual form. In [13] authors show how AToM^3 is used to build visual

modeling environments for manipulating Causal Block Diagrams models to simulate Ordinary Differential Equations and generate textual code for the object-oriented continuous simulation language OOCSMP.

MetaEdit+ is a commercial CASE framework for rapid development and usage of domain-specific visual languages [81]. In MetaEdit+ a domain-specific language first has to be designed and implemented by a domain expert. Then the language can be used by a user without a need to do any programming, problems are stated visually. Code generators have to be defined on a metamodel using a built-in scripting language, which in a rather straightforward way transform models into some external language.

The VLCC (Visual Language Compiler-Compiler) [8] is a grammar-based system for automatic generation of visual programming environments. In VLCC, graphical tools define visual languages to create both graphical objects and composition rules. The underlying grammar formalism in VLCC is the positional grammar model, which is expressive, but also efficient to parse.

Moses [21] is a framework for defining syntax and semantics of DSVLs focused on (but not limited to) the modeling of discrete-event systems. The abstract syntax of visual languages is defined by *attributed graphs* and set of predicates over such graphs to enable validation of structures and attribution. In a similar manner as our implementation, Moses allows embedding program fragments into visual languages to provide better functionality. The semantics of visual languages in Moses is specified in the form of an interpreter using *Abstract State Machines* [27] – a language for formal specification of operation semantics. The framework also has a visual editor used for creating visual notations and visual syntax checking.

2.5 Summary

In this chapter we defined the context of the thesis with respect to the given problem statement. The foundation for our work from the automated program construction standpoint has been set by various works on the structural synthesis of programs. Attribute grammar techniques influenced the thesis in choosing notations and evaluation strategies. Model-based software development paradigm turned out to be

very attractive approach for software engineering where our work fits well. Domain-specific visual languages have proven their usefulness over several decades and the current thesis tries to give precise semantics of such languages for developing software in time-efficient and convenient way.

Chapter 3

Flat languages

Our goal is to define a class of declarative languages that will have well-defined semantic properties, to propose a theoretically sound method of implementation of semantics of these languages, and to test it in practice.

We consider declarative languages where a text can be not only a specification of a single program, but also a description of device or a system (its model) that allows one to ask several different questions about the specified concept. This means that a program can be obtained from a declarative specification and a goal (that is, a problem statement describing what is needed). We have restricted the set of specification languages considered here to structurally very simple languages that we call flat languages.

Flat languages are declarative languages suitable for defining and composing objects into descriptions of concepts by connecting their components using ports. Such languages can be called flat, because they have little syntactic structure. For instance, they are without explicit looping and branching statements (in general, Turing-incomplete). However, these control structures – looping etc, can be themselves added as components. Flat languages allow creating hierarchical descriptions which can be unfolded into flat forms. The meaning of specifications in flat languages is hidden in the types of objects and in the way the objects are connected.

Specification languages in engineering domains (VHDL, SDL, etc) are declarative languages where concepts (entities) are often connected by means of ports, gates or channels. In essence, most of the declarative specification languages are flat languages, although having some features that are difficult to express through the local semantics of objects.

3.1 Types

When designing a language, the issue of choosing an appropriate typing systems is essential. There are two typing systems that can be distinguished: *nominative* (or *nominal*) and *structural* typing [70]. The question is, how do typing systems handle equivalence and subtyping and which one fits our needs. In structural approach, an equality and a subtyping relations depend purely on the structure of types. In nominative approach, these relations depend not only on structure of types but also on explicitly given names (tags) of types.

In the example below, declarations of three types are given:

```
AndGate:(int out, int in1, int in2)
OrGate:(int out, int in1, int in2)
XorGate:(int result, int in1, int in2)
```

In a structural typing system, the three types would be equivalent, as the structure of types is identical. In a nominative typing system, the type `XorGate` would not be equivalent, as the names `out` and `result` are different.

The second example is related to subtyping.

```
UnaryGate:(double out, double in1)
BinaryGate:(double out, double in1, double in2)
```

In a structural typing system, BinaryGate is a subtype of UnaryGate. In a nominative system, to make BinaryGate the subtype of UnaryGate, one has to explicitly declare the subtyping relation (here denoted by the "super" keyword):

```
BinaryGate:(super UnaryGate; double in2)
```

For our language, first, we need a separation of concepts, even if concepts share the same structure, and, second, the subtyping should be explicit. Having such criteria in mind it is obvious that nominative typing fits better, though there is a trade-off for tidiness and elegance.

We define *flat languages* as follows. There is given a finite set of *primitive types* (i.e. value types) and a countable set of *names*.

New *types* are defined by a construction

$$a : (t_1, \ldots, t_n;\ a_1 : s_1, \ldots, a_k : s_k),$$

where a is a name given to a new type and this type is called the *compound type*. The construction $a_i : s_i$ defines a *component* of a, where a_i is a unique name of the component, i.e. $a_i \neq a_j$ for $i \neq j$, and s_i is a name of a known type, $i = 1, ..., k, j = 1, ..., k$. We can refer to a component a_i of the type a as $a.a_i$.

The subtyping relation is defined on types: a type a is a subtype of compound types $t_1, \ldots, t_n$ (the list of supertypes may be empty). Overriding of component names is not allowed, i.e. components of a subtype cannot shadow components of supertypes.

3.2 Classes

Types of objects in a flat language are represented by *classes*. A class includes a compound type as a part of it. A *class* is defined by a construction

$$c : (s, (p_1, \ldots, p_m), \mathfrak{S}),$$

where

- c is the name of a class,
- s is its type, whereas c matches the name of type s and components of a type are also the components of a class.
- $p_1, \ldots, p_m$ are the *ports*. A port p is a connection point of a class representing one of its components or a whole instance of the class. Ports allow instances of classes (objects) to bind together their components. If a_i is a component of class c and port p represents this component, then the name of port p is the name of component a_i.The intuition behind ports (by analogy with object-oriented languages) is that they provide access to class attributes the same way as "get" and "set" methods provide access to the fields of classes e.g. in the Java language.
- $\mathfrak{S}$ is a definition of the local semantics of the class that can be defined only in terms of s, i.e. using only the components of s. Local semantics $\mathfrak{S}$ is not defined here, it depends on a concrete flat language.

3.3 Syntax

A text in a flat language is a specification of an object, a system or a process. *Text* is a sequence of *statements*. The syntax of a flat language is given by a set of classes and by the following simple rules in EBNF [1] as follows:

$$
\begin{aligned}
Text &::= \{Statement;\} \\
Statement &::= ObjectDeclaration | Binding \\
ObjectDeclaration &::= TypeName\ ObjectName \\
TypeName &::= Primitive | ClassName \\
Port &::= ObjectName | ObjectName.ComponentName \\
Binding &::= Port{=}Port
\end{aligned}
$$

Statement is either an object declaration or a binding. Object is an instance of a class. *Binding* is an equality between ports, where a port can be also a whole object.

3.4 Semantics

Semantics of a flat language depends only on the semantics of its classes and on the semantics of bindings. In this section we show only the semantics of bindings. Local semantics of classes will be introduced in Chapter 5. Semantics of bindings is the following:

a) if ports in a binding have one and the same primitive type, then the values of bound components are considered to be equal.

b) if ports in a binding have one and the same compound type, then a binding is recursively defined also for all pairs of the respective components of a type. Example: the binding $p.x = q.y$ for the ports x and y that have the type defined as $a : (a_1 : s_1, \ldots, a_k : s_k)$ defines also the bindings $p.x.a_1 = q.y.a_1, \ldots, p.x.a_k = q.y.a_k$ as well as bindings for all components of the types

[1] For brevity, here and in further sections we use quotation marks for specifying terminal symbols only when it is required to distinguish them from the syntax of EBNF

of $a_1, \ldots, a_k$. If ports have different types, but have a common supertype, then bindings are recursively defined for components of the supertype.

3.5 Examples

Let us look at a small example of a flat language that is for specifying reliability of devices through the reliabilities of their components and the structure of a device. Types are *double*, *Basic*, *Parallel*, *Series*. The type *double* is a primitive type for numbers. The type *Basic* represents components of a device that have a value of reliability, let it be a probability of a correct operation of a device or a component during a given period of time. This probability is represented by a variable p that is of the type *double* and is a component of an object of type *Basic*. Let us introduce also another variable q that expresses a probability of error occurring during the given period of time. The type *Parallel* represents a substructure of a device that is composed of two parts denoted by *part1* and *part2* in such a way that if at least one part works correctly, then the composition works correctly. These parts are of type *Basic*. The type *Series* represents a substructure of a device that is composed of two parts denoted by *part1* and *part2* in such a way that it operates correctly if both parts operate correctly. These parts are of type *Basic*. This is a rather typical example of a small domain-specific language in engineering. There are some computations that can be performed on the objects of type *Parallel* and *Series*. These computations and the notations introduced above are summarized in Table 3.1.

Some words have to be said about the Computations column in Table 3.1. We see equations there, and one may expect that these equations can be used in different ways, not only for computing the value of a variable on the left side of an equation. This column presents a computational semantics of the types to a user not interested in an implementation of the language. The implementation can be made in several ways. In particular, one could extract two assignments from the equation for *Basic*:

p:=1-q

q:=1-p,

or one can use a numeric equation solver for solving the equation. We postpone the

Name of type	Supertype	Components	Computations	Comment
double				primitive numeric type
Basic		*double p* *double q*	$p + q = 1$	
Parallel	*Basic*	*Basic part1* *Basic part2*	$q = part1.q * part2.q$	
Series	*Basic*	*Basic part1* *Basic part2*	$p = part1.p * part2.p$	

Table 3.1: An example of a flat language

discussion of the implementation now and return to it in Section 5.4.

We extend the flat language with the values of primitive types that are handled as predefined objects.

An example written in the language of reliability is as follows:

```
Basic c1, c2, c3;
c1.p=0.99;
c2.p=0.97;
Series sr;
sr.part1=c1;
sr.part2=c2;
Parallel pr;
pr.part1=sr;
pr.part2=c3;
```

This is a description of five objects `c1`, `c2`, `c3`, `sr`, `pr` connected by means of 4 equalities, and assignment of values to primitive objects `c1.p`, `c2.p` that are components of objects `c1` and `c2`. This specification can be presented visually, as soon as one has means to represent objects of types *Basic*, *Parallel*, *Series*, as well as a possibility to connect ports and to introduce values of primitive types. A visual representation of the specification is shown in Figure 3.1. This text (or the scheme in Figure 3.1) can be used as a specification of several computations. A computation is defined as soon as a goal is given (cf. Prolog). A *goal* states the input and the output

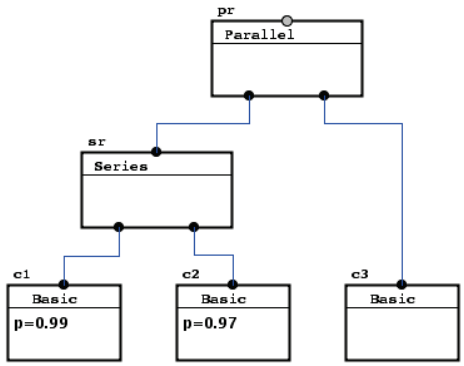


Figure 3.1: Visual specification in a flat language

of a computation. For example, the following two goals are interesting (although there are more meaningful goals):

`c3.p->pr.p` and `pr.p->c3.q`

The first requires finding the resulting value `pr.p` of reliability of the device `pr` depending on the reliability value `c3.p` of the component `c3`, and the second requires finding the probability of error `c3.q` of the component `c3` that gives the given reliability of device `pr`. $\diamond$

The following example demonstrates how attribute grammars can be implemented using a flat language. The syntax of a context free language is given by rules of the form:

$$p_0 \leftarrow w_1 p_1 w_2 p_2 \dots w_k p_k,$$

where $p_0, \dots, p_k$ are nonterminal symbols of the language and $w_1, \dots, w_k$ are possibly empty sequences of terminal symbols. The semantics of the language is given by means of an attribute grammar. First, we introduce a type t_p for each nonterminal symbol p that includes the attributes of the symbol p as its components. Then we introduce types and classes for rules as follows. A type t_r of a rule r has components $c_0, \dots, c_k$ with respective types $tp_0, \dots, tp_k$ for its nonterminal symbols $p_0, \dots, p_k$. A class of a rule r includes besides the type t_r also attribute dependencies as defined in the attribute grammar. By the definition of an attribute grammar, this flat language is sufficient for expressing attribute semantics of any sentence of the given language.

A specification in the flat language, let us call it an attribute model [69], can be built from an abstract syntax tree of a text [15, 92]. This specification can be used for computing the synthesized attribute of the nonterminal symbol representing the whole text. This is a dynamic evaluation of attributes of a language given by an attribute grammar. In order to be able to express attribute semantics of a language by means of one single specification in a flat language and to compose a static attribute evaluation algorithm, we have to extend the types of rules. ⋄

Chapter 4

Attribute models

In this chapter we present attribute models and describe algorithms for performing computations on such models. The semantics of specifications can be given by means of attribute models in three steps. First, a specification has to be translated into an attribute model. Second, given a goal containing some input attributes and a set of target attributes, attribute evaluation is planned, and if it succeeds, it produces an algorithm for computing values of intermediate and target attributes. Third, having an evaluation algorithm, a program in any convenient (for a user) programming language can be extracted for computing the values by executing it. In our approach, all three steps are fully automated and allow to synthesize efficient programs from declarative specifications.

4.1 Simple attribute models

In this section we give definitions of attributes, attribute dependencies and attribute models in a conventional way, not relating them to syntax of a flat language.

Definition 4.1. *Attribute* a^σ is a variable of a type σ.[1]

Definition 4.2. *Simple attribute dependency* is a functional dependency between attributes.

Let us use the following notation for a simple attribute dependency:

$$x_1, ..., x_m \to y_1, ..., y_n\{f\},$$

[1] We often omit type annotations of attributes for brevity.

where f is a function of m arguments $x_1, \ldots, x_m$ computing a value of n-tuple $(y_1, \ldots, y_n)$, i.e. $x_1, \ldots, x_m$ are the inputs and $y_1, \ldots, y_n$ are the outputs of the attribute dependency. We say that the inputs and the outputs are bound by the attribute dependency. It is important to notice that functions realizing attribute dependencies are not defined in terms of attribute models and are assumed to be given from the outside.

Definition 4.3. *Simple attribute model* is a pair $\langle A, R \rangle$, where A is a finite set of attributes and R is a finite set of attribute dependencies binding these attributes.

Definition 4.4. An *equality* $x = y$ of two attributes x and y having the same type or a common supertype is a shorthand notation for two attribute dependencies $x \to y\{id\}$ and $y \to x\{id\}$, where id is a polymorphic identity function. If attributes are not of primitive type, equality implies the equality of their respective components.

Two or more attribute models can be composed into a new attribute model by binding some of their attributes by equalities.

Definition 4.5. For attribute models $M_1 = \langle A_1, R_1 \rangle, \ldots, M_n = \langle A_n, R_n \rangle$ and a set of equalities $S = \{a = b, \ldots, d = e\}$ that bind some attributes of models $M_1, \ldots, M_n$ we denote by $\cup_S(M_1, \ldots, M_n)$ an attribute model with the set of attributes $\bigcup_{i=1}^{n} A_i$ and the set of attribute dependencies $S \cup (\bigcup_{i=1}^{n} R_i)$, and call it a *composition* of $M_1, \ldots, M_n$ with bindings S.

Remark 4.1. When building a composition of attribute models, renaming of attributes may be required. A straightforward way to do it is to add the name of a model where an attribute came from to its name. This introduces composite names, e.g. $m.x, m.y$ for attributes x, y of an original model m.

4.2 Computational problems on attribute models

Definition 4.6. Let U and V be two sets of attributes of an attribute model M. A *computational problem* on the attribute model M is denoted by

$$G_M = U \to V,$$

where U is a set of input attributes (or just inputs) and V is a set of output attributes (or outputs) of a computational problem G_M.

The computational problem states a goal, that is, given values of attributes from U, requires to find values of attributes of V using attribute dependencies of M.

For two computational problems $G = U \to V$ and $G' = U' \to V'$, the computational problem G is greater than the computational problem G' (denoted by $G \succ G'$), if the following condition holds:

$$(U \subset U' \wedge V' \subseteq V) \vee (U \subseteq U' \wedge V' \subset V).$$

The given definition is required for the evaluation algorithm, where, having computed new attributes, computational problems get reduced to smaller problems in comparison to initial ones.

4.3 Attribute evaluation planning

Planning is a term used in the field of artificial intelligence to describe algorithms that define the behavior of agents in a particular environment [43]. In the context of attribute models, planning algorithms determine the computational path starting from some initial attributes to the target attributes.

We describe now a method that for a goal in the form of a computational problem $U \to V$ on a simple attribute model decides (plans) whether there is a way to compute values of attributes of V from given values of attributes of U, and in the case of the positive answer produces an algorithm for solving the computational problem (i.e. computing the values of target attributes).

Definition 4.7. *Value propagation* is a procedure that, for a given attribute model M and a set of attributes $U \subset A$, decides which attributes are computable from U and produces a sequence of attribute dependencies. Constructed sequence of attribute dependencies is an algorithm for evaluating the attributes that are computable on the model M.

Algorithm 4.1 $valprop(U, V, R, algorithm)$

{U is a set of inputs and an accumulator for newly computed attributes}
{V is a set of outputs of a computational problem}
{R is a set of attribute dependencies in the model}
{$algorithm$ is an initially empty list of dependencies}
for all $r \in R$ **do** {for each attribute dependency}
  $setcounter(r,\ \#inputs(r))$ {set counter to the number of inputs}
  if $counter(r) = 0 \wedge outputs(r) \not\subseteq U$ **then** {if not all outputs are computed}
    $U := U \cup outputs(r)$
    $algorithm := add(algorithm,\ r)$
  end if
end for
$K := U$ {the set of computed attributes}
while $\neg done \wedge V \not\subseteq U$ **do** {proceed while not all problem outputs are computed or there are some dependencies remain unvisited}
  $done := true$
  for all $k \in K$ **do**
    for all $r \in relations(k)$ **do** {k is an input for r}
      $setcounter(r,\ counter(r) - 1)$
      if $counter(r) = 0 \wedge outputs(r) \not\subseteq U$ **then** {dependency is picked only if its outputs were not previously computed}
        $N := N \cup outputs(r)$
        $algorithm := add(algorithm,\ r)$
        $done := false$
      end if
    end for
  end for
  $K := N \setminus U$ {for the next iteration, only newly computed attributes are required}
  $U := U \cup N$ {add newly computed attributes into the set of all computed attributes}
end while

The idea for the value propagation planning algorithm that works in linear time originated from the work by Dikovsky [18] and adapted by Tyugu for higher order dataflow schemas [84]. Versions of this algorithm have also been presented in [72,86]. Algorithm 4.1 works step by step as follows. At each step it tries to find an attribute dependency whose inputs are all known (initially given or computed) and some of outputs are not computed. In the positive case, the attribute dependency will be added to the algorithm being built and all its outputs will be added to the set of computed attributes. In the negative case (if there is no such attribute dependency and not all outputs of the problem have been found), the problem is unsolvable. Initially the set of computed attributes equals to the set of given attributes U and the algorithm (i.e., the sequence of attribute dependencies) is empty. The value propagation procedure does not consider concrete values of attributes from the input set U, it just assumes that attributes are computable. For each attribute dependency the value propagation stores a counter indicating the number of unknown inputs. If an attribute is or becomes computable, counter is decreased for each dependency where such attribute is an input attribute. If for an attribute dependency a counter is zero, such dependency can be added to the algorithm and its output attributes become computable. It is easy to notice that due to the usage of counters the algorithm works in linear time, i.e. each edge in the graph is visited only once.

The described method does not give a minimal algorithm for solving a problem in general — the algorithm may include steps that are unnecessary for solving the problem. The procedure of minimizing the algorithm will be discussed in Section 4.6.

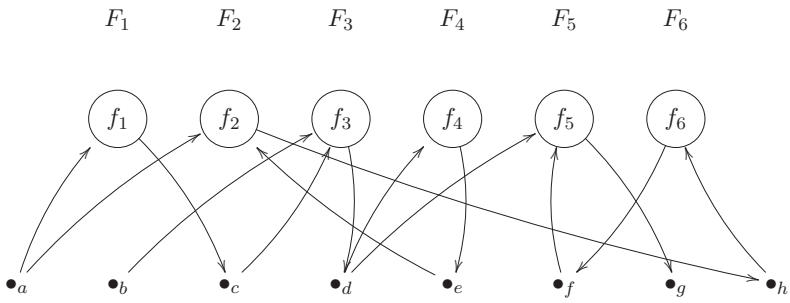


Figure 4.1: Bipartite graph of a simple attribute model

Example. Let us consider the following simple attribute model with eight attributes and six attribute dependencies:

$$\begin{aligned}\langle A = \{a, b, c, d, e, f, g, h\},\\ R = \{a \to c\{f_1\};\\ a, e \to h\{f_2\};\\ b, c \to d\{f_3\};\\ d \to e\{f_4\};\\ d, f \to g\{f_5\};\\ h \to f\{f_6\};\}\rangle\end{aligned}$$

It is always possible to present an attribute model in the form of a directed bipartite graph, where edges correspond to directed bindings between attributes and dependencies. Figure 4.1 shows it for the present example with additional labels $F_1, \ldots, F_6$ for attribute dependencies.

Let us try to solve a computational problem $G_1 = U : \{a, b, f\} \to V : \{g\}$ on a given attribute model. One of the outcomes of the value propagation in this example is a sequence $\{F_1, F_3, F_4, F_2, F_5\}$. From the produced evaluation algorithm it is clear that computed variables e and h are not needed to solve the given computational problem (i.e. to compute an attribute g). The process of minimizing this algorithm will be explained in Section 4.6.

For another computational problem $G_2 = U : \{a, b\} \to V : \{g\}$ that is greater than G_1 (i.e. $G_2 \succ G_1$), an attribute evaluation algorithm produced by the value propagation is a sequence $\{F_1, F_3, F_4, F_2, F_6, F_5\}$. $\diamond$

4.4 Higher-order attribute models

Let A be a set of attributes and P a set of computational problems with inputs and outputs from A.

Definition 4.8. *Higher-order attribute dependency* (*hoad*) is a functional dependency that has inputs from $A \cup P$ and outputs from A.

Higher-order attribute dependencies have the following form:

$$s_1, \ldots, s_k, x_1, \ldots, x_m \to y_1, \ldots, y_n\{f\},$$

where $s_i \in P$ $(i = 1, \ldots, k)$ and s_i is called a subtask. Subtasks are the computational problems that have to be solved in order for a *hoad* to become usable in the attribute evaluation.

Definition 4.9. *State* W of a planning process on an attribute model M is the union of inputs of a computational problem and a set of attributes computable by the planned part of an algorithm.

A subtask $U \to V$ is solvable on an attribute model M in the state W if a problem $U \cup W \to V$ is solvable on M.

Definition 4.10. *Higher-order attribute model* is a pair $\langle A, R \cup R^{ho} \rangle$ where A is a set of attributes, R is a set of simple attribute dependencies and R^{ho} is a set of higher-order attribute dependencies.

This extension makes a big difference in the following: higher-order attribute models have expressive power that enables to synthesize recursive, branching and cyclic programs where respective control structures, i.e. recursion, branching and loops are preprogrammed and represented as higher-order attribute dependencies. Detecting the solvability of a problem and synthesizing an algorithm on a higher-order attribute model has exponential time complexity with respect to the number of higher-order dependencies (see the remark in Section 4.5).

4.5 Evaluation of higher-order attributes

Let us distinguish two cases of higher-order attribute models: in the first case a model contains a single higher-order attribute dependency (*hoad*) and in the second case it has more than one *hoad*. The evaluation strategy is quite obvious in the first case: first use only simple attribute dependencies and at the end use the higher-order one. Thereafter, if still needed, use simple attribute dependencies again. Time complexity of the search remains linear in this case with respect to size of an attribute model.

In the second (general) case, when an attribute model M contains several higher-order attribute dependencies, the strategy for construction of an evaluation algorithm is as follows.

First, the procedure of simple value propagation is invoked using only attribute dependencies that are not higher-order. If this does not solve the initial computational problem $U \rightarrow V$ (does not give values of all outputs of the problem), a *hoad* is applied, if it is applicable. A *hoad* is applicable in a state of a planning process if and only if all its inputs are given (i.e. are present in the state) and all its subtasks are solvable and it computes values of some attributes that have not been evaluated yet. A sequence of applicable attribute dependencies obtained in this way is called *maximal linear branch* (*mlb*). It contains at most one *hoad* at the end of the sequence. There are three possible outcomes of the procedure of finding a *mlb*:

1. After constructing a *mlb* the problem is solvable.

2. A *mlb* cannot be found and the problem is unsolvable.

3. A *mlb* can be found and the initial problem $U \rightarrow V$ is reduced to a smaller problem $U' \rightarrow V'$, where $U' = U \cup Y$, $V' = V \setminus Y$, and Y is the set of outputs of a *hoad* used in *mlb*.

This procedure (construction of *mlb*) is repeatedly applied until the problem is solved or no more *mlb*s can be constructed.

It is important to notice that for applying a *hoad* we have to solve all its subtasks. This means that the whole procedure of problem solving must be applied for every subtask. This requires a search on an *and-or* tree of subtasks on the attribute model. The root of a tree corresponds to the initial problem, and it is an *or*-node, because there may be several possible *mlb*s for this problem. *And*-nodes correspond to higher-order attribute dependencies and have one successor for its each subtask, plus one successor for the reduced task that has to be solved after applying the *mlb*. *Or*-nodes of the tree correspond to the subtasks that have to be solved for their parent *and*-node. The search on the *and-or* tree is depth-first search with backtracking.

Algorithm 4.2 $mlbsearch(W,\ R,\ R^{ho},\ algorithm,\ path)$

```
{W is a state (set of computed attributes)}
{R, R^ho are sets of simple and higher-order dependencies}
{algorithm and path are lists of dependencies}
MLB:
while R_s ≠ ∅ do
  OR:
  for all r ∈ {r' | r' ∈ R_s ∧ counter(r') = 0 ∧ last(path) ≠ r'} do
    allsolved := true
    add(path, r)
    AND:
    for all s ∈ subtasks(r) do
      solved = valprop(W ∪ inputs(s), outputs(s), algorithm_s)
      if ¬solved then
        mlbsearch(R, R^ho, algorithm_s, path)
        solved = valprop(W ∪ inputs(s), outputs(s), algorithm_s)
        allsolved := allsolved ∧ solved
        if ¬allsolved then
          continue OR
        end if
      end if
      cache(algorithm_s, s)
    end for
    if allsolved then
      algorithm := add(algorithm, r)
      W := W ∪ outputs(r)
      R_s := R_s \ {r}
      continue MLB
    end if
  end for
  break MLB
end while
```

The planning algorithm for evaluating the attributes on higher-order attribute models is shown in Algorithm 4.2.

Remark 4.2. It is useful to notice that specifications of attribute dependencies (and higher-order attribute dependencies) can be considered as propositional formulas where arrows denote implications and commas denote conjunctions. Building an attribute evaluation algorithm for a particular computational problem with inputs $u_1, \ldots, u_m$ and outputs $v_1, \ldots, v_n$ corresponds then to a derivation of the formula $u_1 \wedge \ldots \wedge u_m \supset v_1 \wedge \ldots \wedge v_n$ in the intuitionistic propositional calculus (IPC). This correspondence is known as the Curry-Howard isomorphism [30]. We would like to point out here that the evaluation of attributes considered in the present section is an efficient algorithm of program construction in propositional logic programming [61]. This approach gives also an algorithm of proof search for IPC, although some transformation of propositional formulas to the suitable form will be needed in the general case [51]. The proof search for IPC is PSPACE-complete [78], hence the higher-order attribute evaluation also has exponential time complexity.

Example. In this example we will show the problem solving on higher-order attribute models. Let us consider an attribute model with the following attribute dependencies:

$$[y \to a] \to b$$
$$[a \to b] \to x$$
$$x, y \to a$$

and a goal in the form of a computational problem with no inputs and one output $\{\to b\}$ on this model. In fact, solutions (there can be more than one) of this problem are equivalent to the derivation of a so-called Kripke's formula $((((A \to B) \to A) \to A) \to B) \to B$ which is an intuitionistic analog of the classically valid formula $((A \to B) \to A) \to A$ known as the Peirce's law. The idea of encoding arbitrary intuitionistic propositional formulae into sets of formulas with at most one subimplication on the left hand side of a outermost implication is described in [51].

For brevity, in this example we omit implementations of attribute dependencies

and label dependencies as follows:

$$S_1 : [y \to a]$$
$$S_2 : [a \to b]$$
$$R_1 : S_1 \to b$$
$$R_2 : S_2 \to x$$
$$F_1 : x, y \to a,$$

where S_1 and S_2 are subtasks, R_1 and R_2 are higher-order attribute dependencies and F_1 is a simple attribute dependency.

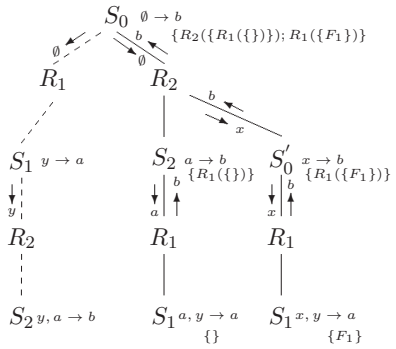


Figure 4.2: Decorated and-or search tree of the example

Figure 4.2 shows a complete and-or search tree for a solution of the computational problem. Let us explain the tree traversal step by step. The root of a tree S_0 is a top-level problem with a goal $\emptyset \to b$ with an empty set of inputs. First, value propagation is applied on S_0 returning an empty sequence of attribute dependencies, because the initial problem has no inputs. In order to find a *mlb*, a higher-order attribute dependency has to be considered. Thus R_1 is picked (first *hoad* in the model). The only input of R_1 is a subtask S_1 that has to be solved. S_1 has one input y and one output a. Again, value propagation is applied in the context of S_1 returning an empty sequence. In order to complete an *mlb* of S_1, a *hoad* has to be used again. R_1 is ignored, because it has just been tried, thus R_2 is picked. Subtask

S_2 of R_2 has an input a and an output b. In addition, y from S_1 is also available. S_2 cannot be solved, because value propagation returns an empty sequence (cannot compute anything from y and a) and there are no other *hoad*s that can be used, consequently, *mlb* of S_2 cannot be constructed. We do backtracking to the node S_1 and then to the S_0 and mark the backtracked branch by dashed lines. Back in S_0, second *hoad* R_2 is chosen. S_2 cannot be solved by value propagation, thus R_1 is used. S_1 is solved trivially (i.e. *mlb* of S_1 is empty), because an input a from S_2 maps to the output a of S_1. *Mlb* of S_2 is successfully constructed and its sequence contains one attribute dependency R_1. The output of R_1 is b and it is exactly what is needed to solve S_2. The subtask of R_2 is solved and an output x of R_2 becomes available. Here is a tricky part. First *mlb* of S_0 is constructed with the following sequence of attribute dependencies: $\{R_2(\{R_1(\{\})\})\}$. Initial problem $S_0 : \emptyset \rightarrow b$ is not solved. Knowing x, S_0 is reduced to a new computational problem $S_0^{'} : x \rightarrow b$. Again, value propagation is applied on $S_0^{'}$ returning an empty sequence. Then, R_1 is picked. In the context of S_1, x and y are known. The former is propagated from $S_0^{'}$ and the latter is an input of S_1. The required output is a. Value propagation returns a sequence with one simple attribute dependency $\{F_1\}$, i.e. a becomes known after applying F_1 with inputs x and y. The subtask S_1 is solved. The output of R_1 is b. After using R_1, the problem $S_0^{'}$ is solved, consequently S_0 is solved. Finally, *mlb* of $S_0^{'}$ is glued together with *mlb* of S_0 and the full algorithm of computing the goal b is as follows: $\{R_2(\{R_1(\{\})\}); R_1(\{F_1\})\}$. $\diamond$

4.6 Optimization

A procedure that gives a minimal algorithm for solving a computational problem has been briefly described in [51] and explained in more detail in Section 4.3.4 of [85]. In the context of this work we call finding a minimal sequence of attribute dependencies for solving a computational problem an *optimization* and extend this procedure to work on higher-order attribute models.

For each attribute dependency included in an evaluation algorithm, the optimization procedure (see Algorithm 4.3) checks, whether the computability of outputs of a problem depend on the application of a particular attribute dependency. If not,

an attribute dependency is excluded from the algorithm. Starting with the set of output attributes of a computational problem, the procedure runs backwards from the end of an algorithm. If one or more outputs of an attribute dependency are in the set of goal attributes, dependency is kept in the algorithm and its inputs are added into the set, otherwise dependency is removed. In the case of subtasks, the optimization procedure recursively traverses the tree of subtasks and add relevant attributes into the set of goals also reducing algorithms of corresponding subtasks if needed. The optimization procedure has linear time complexity.

Example. We continue with example started in Section 4.3. The evaluation algorithm $\{F_1, F_3, F_4, F_2, F_5\}$ for solving the computational problem G_1 is obviously not minimal, i.e. some attribute dependencies (F_4, F_2) are not required for computing the output g. Once the optimization procedure is applied, the algorithm is reduced to $\{F_1, F_3, F_5\}$.

On the other hand, algorithm for solving the larger problem G_2 requires all steps produced by the value propagation from computing g. $\diamond$

4.7 Summary

In the current chapter attribute models are described. Evaluation of attributes on simple attribute models is done using the value propagation technique. Maximal linear branches are introduced for attribute evaluation on higher-order attribute models. The optimization procedure is presented for pruning unnecessary branches and achieving more efficient goal-oriented evaluation algorithms.

Algorithm 4.3 *optimize*$(G,\ algorithm)$

for all $r \in reverse(algorithm)$ **do** {traverse the list backwards}
  $keep := false$
  for all $v \in outputs(r)$ **do**
    if $v \in G$ **then** {keep the relation if one of its outputs is in the set of goals}
      $keep := true$
      $G := G \setminus \{v\}$
    end if
  end for
  if *keep* **then** {add inputs into the set of goals}
    $G := G \cup inputs(r)$
    for all $s \in subtasks(r)$ **do** {add subtask's outputs into the set of goals and recursively optimize subtask's algorithm}
      $G_s := outputs(s)$
      $optimize(G_s, algorithm_s)$
      $G := G \cup G_s$
    end for
  else
    $algorithm := remove(r,\ algorithm)$
  end if
end for

Chapter 5

The specification language

In this chapter we introduce our version of a flat language, we call it a *specification language*. The specification language consists of two layers. The first layer is a *core language* presented in the following section. The core language includes statements for specifying functional dependencies, inheritance, etc. The semantics of specifications in the core language is a translation into attribute models. The second layer constitutes *extensions*, that is, additional statements translatable into the statements of the core language. In Section 5.6 we present a full-scale example in the specification language.

5.1 Core language

The specification language that can be used for two purposes:

- As a program specification together with a goal.
- For specifying a new type.

In this section we present syntax of a subset of the specification language called the *core language*. This language includes more features than presented in the example given in Chapter 3. The main difference is the usage of programs as implementations of functional types. We assume that a concrete implementation of the core language is based on some programming language that we call a *base language*. We have an implementation of flat languages with Java as the base language, see Section 6. However in this chapter we are not going to discuss the implementation details and focus on syntax and semantics of the core language.

Types in the core language are primitive types, compound types and functional types:

- a *primitive type* is any type of the base language (including reference types of Java in our implementation).
- a *compound type* is introduced by writing its specification in the core language. In the general terms of flat languages, a compound type is included in a class. In the core language, all components of a compound type are ports and semantics $\mathfrak{S}$ of a class is represented by functional dependencies.
- a *functional type* is a part of a functional dependency as defined below.

Specifications in the core language are written using five kinds of statements in the following syntax:

$Spec ::= \texttt{specification}\ CName\ [\texttt{super}\ Inh^{+}]\ '\{'((Decl|Bind|Val|FuncDep);)^{*'}\}'$

Specifications start with a `specification` keyword, followed by a class name (*CName*) and a list of superclasses starting with a keyword `super` and separated by commas. The rest of the statements are enclosed between curly brackets.

1. *Declaration of object*:

   $Decl ::= Type\ Id[, Id]$

   $Type ::= \texttt{any}|PrimitiveType|CName$

This declaration specifies an object with a given $Type$, and its name given by the identifier Id. The object as well as its components are called *variables*, and they can be bound by functional dependencies. (A component `b` of an object `a` has a compound name `a.b`.)

If `any` is written instead of a type, then type of the object remains undefined, and it must be determined by some binding in a specification later, i.e. `any` is a type variable.

Example. As the running example in this chapter we are going to use the specification of alternating current circuits. In Section 5.6 the full specification will

be demonstrated. Specification of a complex number that will be used for defining alternating current has the following declaration of variables:

```
specification Complex {
    double re, im, mod, arg;
    ...
}
```

◇

2. *Binding of variables*:

   $Bind ::= Var = Var$

   $Var \ ::= Id[.Var]$

where variables must have the same type or a common supertype. A binding $x = y$ denotes a possibility to compute a value of a variable (x or y) in the case a value of another variable is known. A binding $x = y$ is extended to the respective components of x and y, i.e. if x and y have a component a then $x.a = y.a$ is introduced.

Example. Branch is a specification of a basic concept of circuits and a base class for all circuit elements (Resistor, Capacitor, etc). Branch contains a declaration of four variables of type Complex and an instance of the Ohm's law followed by the corresponding bindings. The rest of the specification will be presented later.

```
specification Branch {
    Complex z, i, u, g;
    Ohm law;
    law.i = i;
    law.z = z;
    law.u = u;
    ...
}
```

◇

3. *Valuation*:

   $Val ::= Var = Const$

where $Const$ is an object of a primitive type.

Example. The specification of Branch also contains a variable PI to which the value of a constant of primitive type is assigned.

```
specification Branch {
    ...
    double PI = Math.PI;
}
```

◇

4. *Functional dependency*:

$$
\begin{aligned}
FuncDep &::= (SFuncDep|HOFuncDep)'\{'Impl'\}'\\
SFuncDep &::= [VarList] \rightarrow Var\\
HOFuncDep &::= Subtask[,Subtask]^*[,VarList] \rightarrow Var\\
Subtask &::= '['VarList \rightarrow VarList']'\\
VarList &::= Var[','Var]^*
\end{aligned}
$$

where the statement *SFuncDep* has one arrow and specifies a functional dependency that represents computing a value of the variable on the right side of the arrow (the result) from given values of variables on the left side (arguments). The statement *HOFuncDep* has several arrows and specifies a higher-order functional dependency (*hofd*). In this case there are also arguments that have a functional type. These are the arguments in square brackets. These arguments are called *subtasks*.

Impl is a name of a program (a Java method in our implementation). This program is an implementation of the functional dependency specified by this statement. The type given by a functional dependency must be the same structural type of the program given by its corresponding implementation, in particular, in the case of a functional dependency with functional arguments the program must also have procedural parameters of appropriate types.

Example. We can specify the Ohm's law by means of functional dependencies, where corresponding implementations will contain arithmetic expressions for calculating correct values.

```
specification Ohm {
    u, i -> r {resistance};
    r, i -> u {voltage};
    u, r -> i {current};
}
```

◇

Example. A higher-order functional dependency is used to specify a tabulating procedure. I.e. if we need to calculate a circuit impedance depending on different frequencies, a loop is required. We can introduce a *hofd* specifically for this purpose with an implementation *tabulate* containing the realization of a loop:

```
[freq -> imp], min, max, step -> success {tabulate};
```

In general, tabulating procedures are specified as *hofd*s, where inputs and outputs of a subtask can be defined depending on a particular problem to solve. This is achieved using binding of inputs and outputs of a subtask with other variables in a specification. ◇

5. *Inheritance*:

   $Inherit ::= CName['\,' \ CName]^*$

   This statement defines an inheritance relation. Overriding variables in subclasses is not allowed to avoid collisions of names.

   Example. Branch is a base class of all circuit elements, including Capacitor, Inductor, Resistor and also classes specifying connections – Series and Parallel, e.g.:

```
specification Capacitor super Branch {
    ...
}
specification Inductor super Branch {
    ...
}
...
```

◇

5.2 Semantics

This section briefly presents the semantic rules that translate statements of the core language into an attribute model. We employ the following definitions to simplify the usage of semantic rules:

- $Var(C)$ – returns a set of names of components of a class C.

- $Var(C)_x$ – returns a set of names of components of a class C adding a prefix "x." to each name.
- $Rel(C)$ – returns a set of relations of a class C.
- $Rel(C)_x$ – returns a set of relations of a class C and renames all variables occurring in a relation by adding a prefix "x.".
- $Dom(x)$ – the domain of a variable x
- $Common(x, y)$ – returns the *least* common supertype of two composite types of variables x and y.
- $\bar{x}$ – denotes a list, where $\bar{x} = x_1, \ldots, x_n$.

Inference rules

We introduce two big-step inference rules. The rule INFR takes a list of statements $xs; x$, where x is the last element of the list and xs is the rest, and applies a semantic rule for x relying on the information already derived from xs. If the list of statements is empty, the rule EMPTY is used.

$$\textsc{Empty}\ \frac{}{() \rightsquigarrow \langle \emptyset,\ \emptyset \rangle} \qquad\qquad \textsc{Infr}\ \frac{xs \rightsquigarrow \langle A, R \rangle_M \qquad x \xrightarrow{M} \langle A^{'}, R^{'} \rangle}{(xs; x) \rightsquigarrow \langle A \cup A^{'},\ R \cup R^{'} \rangle}$$

$\langle A, R \rangle_M$ denotes an attribute model with the name M. $x \xrightarrow{M} \langle A^{'}, R^{'} \rangle$ means that from the statement x and the model M a new model $\langle A^{'},\ R^{'} \rangle$ is obtained the name of which we are not interested in.

It is important to notice that classes are defined globally, so there is no need to carry the information about types in the rules.

Declarations of variables

A declaration of variables is handled by three rules PRIM, ANY and COMP for the variables of primitive, any and compound types respectively. The rule VARLIST is for multiple variables declared in one line with the same type. Rules PRIM and ANY simply add new attributes into the model. The rule COMP first adds a new attribute x and all attributes corresponding to the components of a class C rewriting their names by prefixing "x.". Second, the rule rewrites all functional dependencies

with new attribute names and adds new attribute dependencies $x.c_1, \ldots, x.c_k \rightarrow x,\ x \rightarrow x.c_1,\ \ldots,\ x \rightarrow x.c_k$ meaning that if all components of x are computable, then x is also computable, and vice-versa, any component of x can be computed from x if x is computable.

$$\textsc{Prim}\ \frac{}{p\ x \xrightarrow{M} \langle \{x\},\ \emptyset \rangle}\ (x \notin A_M) \qquad \textsc{Any}\ \frac{}{\texttt{any}\ x \xrightarrow{M} \langle \{x\},\ \emptyset \rangle}\ (x \notin A_M)$$

$$\textsc{Comp}\ \frac{}{\begin{array}{l} C\ x \xrightarrow{M} \langle \{x\} \cup Var(C)_x, \\ \quad Rel(C)_x \cup \{x.c_1, \ldots, x.c_k \rightarrow x, \\ \qquad x \rightarrow x.c_1,\ \ldots,\ x \rightarrow x.c_k\} \rangle \end{array}}\ (x \notin A_M,\ c_i \in Var(C), i = 1, \ldots, k)$$

$$\textsc{VarList}\ \frac{t\ x_i \xrightarrow{M} \langle A_i,\ R_i \rangle,\ i = 1, \ldots, n}{t\ x_1, \ldots, x_n \xrightarrow{M} \langle \bigcup_{i=1}^{n} A_i,\ \bigcup_{i=1}^{n} R_i \rangle}$$

Bindings of variables

The rule BINDP handles the binding of variables of primitive types. It adds two attribute dependencies into the model where realizations are the assignments to the corresponding variables.

For the binding of two variables of compound type, the rule BINDC is used. First, it uses the function *Common* to derive the least common supertype C. Second, for each component c_i of C, a new binding is created between components c_i of x and y.

$$\textsc{BindP}\ \frac{}{x = y \xrightarrow{M} \langle \emptyset,\ \{x \rightarrow y\{y := x\}, y \rightarrow x\{x := y\}\} \rangle}\ (x, y \in A_M,\ x, y \text{ ARE PRIMITIVE})$$

$$\textsc{BindC}\ \frac{\begin{array}{c} C = Common(x, y) \\ c_i \in Var(C)\ (x.c_i = y.c_i) \xrightarrow{M} \langle \emptyset, R_i \rangle,\ i = 1, \ldots, n \end{array}}{x = y \xrightarrow{M} \langle \emptyset,\ \{x \rightarrow y\{y := x\}, y \rightarrow x\{x := y\}\} \cup \bigcup_{i=1}^{n} R_i \rangle}\ (x, y \in A_M,\ x, y \text{ ARE COMPOSITE})$$

Valuations

The valuation is a special case of binding. It is represented by an attribute dependency with no inputs and an output x. The realization of this dependency is an assignment of c to x.

$$\textsc{Const}\ \frac{}{x = c \xrightarrow{M} \langle \emptyset,\ \{\rightarrow x\{x := c\}\}\rangle}\ (x \in A_M, c \in Dom_{prim}(x))$$

Functional dependencies

The translation of functional dependencies into attribute dependencies is straightforward.

$$\textsc{SFD}\ \frac{}{x_1, \ldots, x_n \rightarrow y\{m\} \xrightarrow{M} \langle \emptyset,\ x_1, \ldots, x_n \rightarrow y\{m\}\rangle}\ (x_1, \ldots, x_n, y \in A_M)$$

$$\textsc{HOFD}\ \frac{}{\bar{S}, \bar{x} \rightarrow y\{m\} \xrightarrow{M} \langle \emptyset,\ \bar{S}, \bar{x} \rightarrow y\{m\}\rangle}\ (S_i = [\bar{a} \rightarrow \bar{b}], \text{ and } \bar{x}, y, \bar{a}, \bar{b} \in A_M)$$

Declarations of superclasses

The rule Inherit provides an attribute model of superclass C which is merged into attribute model M of a subclass using the inference rule.

$$\textsc{Inherit}\ \frac{}{\texttt{super}\ C \xrightarrow{M} \langle Var(C),\ Rel(C)\rangle}$$

5.3 Specifying computational problems

The following statement allows one to ask various questions about a specified concept:

$$Goal ::= VarList\ \rightarrow\ VarList.$$

In other words, *Goal* is a specification of a computational problem for attribute evaluation (see Section 4.2). List of variables on the left-hand side of the arrow are inputs and on the right-hand side are outputs of a goal. It is assumed that values of inputs are given from the outside, i.e. as arguments to a program generated from a specification. An evaluation algorithm is optimized according to the outputs of a goal (see Section 4.6).

5.4 Extension of the core language

There is a standard extension of the core language that can be easily translated in the latter by presenting new statements of the language as collections of statements of the core language.

The standard extension of the core language includes the following new statements:

1. *Alias*

   $Alias$ ::= `alias` $['('Type')']$ Id $'=('VarList')'$

Alias defines a new variable with the name Id. This variable is a tuple of variables listed in the parentheses. $Type$ is an optional parameter. If it is given, it explicitly states that all elements of an alias should be of the given type. A binding of two aliases means the binding of their respective elements, with the restriction that the number of elements should be equal and types of elements in each binding should match (have a common supertype).

Example. The binding `x = y` in the specification below yields additional two binding `a = c` and `b = d`.

```
int a, b, c, d;
alias x = (a, b);
alias y = (c, d);
x = y;
```

◇

It is also possible to declare an alias in one statement and assign a list of its components using the second statement. But assigning a list of variables can be done only once in the specification (the structure of aliases cannot be changed).

$$AliasDecl ::= \texttt{alias}\ ['('Type')']\ Id$$

$$AliasAssign ::= Id\ '=['VarList']'$$

For every alias declared in a specification, a special constant of type `int` is automatically added providing a number of elements of an alias.

Example. *Temp* is a concept of a temperature with a variable *t*. The following specification contains three instances of *Temp*. This example demonstrates how to use a *length* constant to distribute a value among several variables.

```
Temp t1, t2, t3;
alias (double) tempr = ( t1.t, t2.t, t3.t );
tempr.length -> tempr {init};
```

The realization of a functional dependency (*init*) is used to distribute initial temperatures to the variables t by means of the alias *tempr* taking into account the length of *tempr*. The following statements are implicit in the specification:

```
int tempr.length;
tempr.length = 3;
```

◇

2. *Alias with a wildcard*

$$AliasW ::= \texttt{alias}\ [\,'('Type')'\,]\ Id\ '=('\ Wildcard\ ')'$$
$$Wildcard ::= *.Id$$

Alias with a wildcard is a variable whose list of components depends on the particular specification where such statement occurs. This list includes all variables of the components defined in the same specification and names of such variables are equal to the identifier specified in $Wildcard$. The order of components in the list is not predefined, but it remains fixed during the computations. This alias is used for distributing and collecting some values for all objects defined in a specification.

Example. The purpose of this example is to show the difference between declaring aliases with explicit type and without.

```
specification A {
   int x;
}
specification B {
   String x;
}
specification Example {
   A z;
   B u, v;
   alias (String) s = (*.x);
   alias r = (*.x);
}
```

The alias s contains the list of two elements $(u.x, v.x)$ while the alias r contains three elements $(z.x, u.x, v.x)$. $\diamond$

3. *Accessing alias elements by indices*

Most of the programming languages that support tuples have functions (e.g. fst, snd) for accessing elements of tuples by their position from left to right. In our flat language we define the possibility to refer to a variable in a tuple by its position number starting with zero. In order to achieve this we need to extend the grammar for Var:

$$Var \ ::= Id[.Var] \mid Id.\#[.Var],$$

where # is a position number.

Example. This example demonstrates the feature. The binding `z.0 = z.1` yields the binding `x = y`.

```
alias z = (x, y);
z.0 = z.1;
```

If x and y are also aliases, it is possible to refer to their components as well (`z.0.1 = z.1.0` produces `b = c`):

```
alias x = (a, b);
alias y = (c, d);
alias z = (x, y);
z.0.1 = z.1.0;
```

$\diamond$

If a variable x has a compound type C containing, for instance, a component a, and if x is the first element of an alias z, it is also possible to reference a component $x.a$ through z as $z.0.a$.

4. *Accessing alias elements with a wildcard*

$$TmpAlias \ ::= Var'.*.'['('Type')']('\#|Id)$$

This extension allows to construct and use temporary tuples from existing aliases. This is demonstrated with an example.

Example.

```
specification A {
   int x;
}
specification B {
   String x;
}
specification Example {
   A u, v;
   B t;
   alias z = (t, u, v);
   -> z.*.x {foo}; //(t.x, u.x, v.x)
   -> z.*.(int)x {bar}; //(u.x, v.x)
}
```

To use such temporary tuples, elements of an alias should be either aliases or have compound types. In the former case elements are accessed by position numbers, in the latter case by component names. $\diamond$

5. *Equations*

$$Equation ::= AExpr \ = \ AExpr$$

Equation is a shorthand for a set of functional dependencies that can be derived from it. For example, `I = U*R;` will denote three functional dependencies: $I, U \rightarrow R\{f_1\}$; $U, R \rightarrow I\{f_2\}$; $I, R \rightarrow U\{f_3\}$; with the corresponding implementations derivable from the given equation: f_1 : `R` = `I/U`; f_2 : `I` = `U` $*$ `R`; f_3 : `U` = `I/R`. We keep open the class of arithmetic expressions that can be used in equations, because this depends on the power of an equation solver for a particular flat language. (In our implementation based on Java [23], the equations are solved analytically and are restricted in such a way that they cannot include occurrences of one and the same variable simultaneously on both sides of an equation. Besides arithmetic operations, an equation may include only functions implemented in `java.lang.Math` class.)

5.5 Semantics of extensions

Extensions are translatable into the statements of the core language. However, in order to give the semantics of the whole specification language in a uniform way, in

this section we present the rules for direct translation of extensions into attribute models.

Aliases

For an alias declaration without a list of components, two variables x and $x.length$ are created.

$$\textsc{AliasEmpty}\ \frac{}{\texttt{alias}\ (t)\ x \xrightarrow{M} \langle \{x,\ x.length\},\ \emptyset \rangle}\ (x \notin A)$$

For a given list of components, the following attribute dependencies are created:

- a dependency $\rightarrow x.length\{x.length := n\}$ that provides length, that is, a number of components bould by an alias;
- a dependency $a_1, \ldots, a_n \rightarrow x$ that computes an alias x if all its components are computed;
- a set of dependencies $x \rightarrow a_1,\ \ldots,\ x \rightarrow a_n$ that compute components of x from x.

$$\textsc{AliasDecl}\ \frac{}{\begin{array}{c}\texttt{alias}\ (t)\ x = (a_1, \ldots, a_n) \xrightarrow{M} \langle \{x,\ x.length\}, \\ \{\rightarrow x.length\{x.length := n\}, a_1, \ldots, a_n \rightarrow x,\ x \rightarrow a_1,\ \ldots,\ x \rightarrow a_n\}\rangle\end{array}}\ (x \notin A,\ \forall a_i (t \equiv type(a_i)))$$

$$\textsc{AliasAssign}\ \frac{}{\begin{array}{c}x = [a_1, \ldots, a_n] \xrightarrow{M} \langle \emptyset, \\ \{\rightarrow x.length\{x.length := n\}, a_1, \ldots, a_n \rightarrow x,\ x \rightarrow a_1,\ \ldots,\ x \rightarrow a_n\}\rangle\end{array}}\ (x \in A,\ a_1, \ldots, a_n \rightarrow x \notin R)$$

Wildcards

In case of a wildcard, a function $components(a)_t$ returns a list of variables of the components of compound types declared on the same specification that have a name a and, optionally, a type t.

$$\textsc{Wildcard}\ \frac{\bar{l} = components(a)_t \qquad R^{'} = \{x \rightarrow a^{'} \mid a^{'} \in \bar{l}\}}{\texttt{alias}\ (t)\ x = (*.a) \xrightarrow{M} \langle \{x\},\ \{\bar{l} \rightarrow x\} \cup R^{'} \rangle}\ (x \notin A)$$

Equations

For an equation, $vars$ returns a set of variables that must exist in A and $solve$ produces a set of attribute dependencies where each variable is an output of a dependency in a set $R^{'}$.

$$\textsc{Equation}\ \frac{xs = vars(aexpr) \qquad R^{'} = solve(aexpr)}{aexpr \xrightarrow{M} \langle \emptyset,\ R^{'} \rangle}\ (xs \subseteq A)$$

5.6 Example: electrical circuits

5.6.1 Definitions

The goal of this example section is to demonstrate the use of the extended core language for specifying a problem domain which has a precise mathematical description – alternating current circuits. For describing a circuit we will need complex numbers for representing the current, voltage, impedance and conductivity. The specification of a complex type is as follows:

```
specification Complex {
    double re, im, arg, mod;
    mod^2 = re^2 + im^2;
    im = mod * sin(arg);
}
```

where the relations between attributes are given by equations instead of functional dependencies.

The basic concept for building a circuit is a branch. It is a superclass for all circuit elements. Complex variables z, i, u, g are the impedance, current, voltage and conductivity respectively. Frequency is f. The complex Ohm's law is specified by the first two equations and the last two equations specify the relation between impedance and conductivity. (The constant PI is going to be used in the specifications below, Math.PI is a constant from Java API.)

```
specification Branch {
    Complex z, i, u, g;
    double f;
    u.mod = i.mod * z.mod;
    u.arg = i.arg + z.arg;
    g.mod * z.mod = 1;
    g.arg + z.arg = 0;
    double PI = Math.PI;
}
```

The specifications of capacitor, inductor and resistor are as follows:

```
specification Capacitor super Branch {
    double omega, C;
    g.re = 0;
    g.im = omega * C;
    omega = 2 * PI * f;
}
```

```
specification Inductor super Branch {
    double omega, L;
    z.re = 0;
    z.im = omega * L;
    omega = 2 * PI * f;
}
```

```
specification Resistor super Branch {
    double r;
    z.re = r;
    z.im = 0;
}
```

Types of series and parallel connections of elements contain two components ($x1, x2$) that can be bound with any variable of a subtype of the branch type.

```
specification Ser super Branch {
    Branch x1, x2;
    z.re = x1.z.re + x2.z.re;
    z.im = x1.z.im + x2.z.im;
    i = x1.i;
    i = x2.i;
    f = x1.f;
    f = x2.f;
}
```

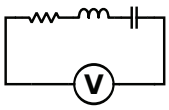
```
specification Par super Branch {
    Branch x1, x2;
    g.re = x1.g.re + x2.g.re;
    g.im = x1.g.im + x2.g.im;
    u = x1.u;
    u = x2.u;
    f = x1.f;
    f = x2.f;
}
```

5.6.2 Specifying a computational problem

Having specified all basic elements of circuits, it is possible to specify, for example, a RLC circuit (see Figure 5.1) and try to solve computational problems on it.

```
specification RLCseries {
    Capacitor cap;
    cap.C = 0.0001;
    Inductor ind;
    ind.L = 0.1;
    Ser ser1;
    ser1.i.re = 1;
    ser1.i.arg = 0;
    Resistor res;
    res.r = 0.5;
    Ser ser2;
    ser1.x1 = cap;
    ser1.x2 = ind;
    ser1 = ser2.x2;
    ser2.x1 = res;
}
```

(a) Specification of RLC circuit



(b) Visual representation

Figure 5.1: Specification of a RLC circuit

One computational problem could be to compute a value of impedance `ser2.z.mod` having provided a frequency of a current. The specification of RLC can be extended to include a goal and statements for setting and distributing a frequency among circuit components.

```
specification Example1 extends RLCseries {
    double f;
    alias (double) fs = (*.f);
    fs.length, f -> fs {distr};
    f -> ser2.z.mod;
}
```

Alias `fs` is constructed using a wildcard to include variables `f` of the underlying components of a circuit. The alternative way is to explicitly add elements to an alias (however, the order might be different):

```
alias (double) fs = (cap.f, ind.f, ser1.f, res.f, ser2.f);
```

The goal of the computational problem is specified by the statement `f -> ser2.z.mod`. Given a value to the variable `f` and invoking the planning procedure, an attribute evaluation algorithm is the following:

```
...
fs_LENGTH = 5;
double[] alias_fs_936  = distr(fs_LENGTH, f);
```

```
cap.f = ((java.lang.Double)alias_fs_936[0]).doubleValue();
ind.f = ((java.lang.Double)alias_fs_936[1]).doubleValue();
ser1.f = ((java.lang.Double)alias_fs_936[2]).doubleValue();
res.f = ((java.lang.Double)alias_fs_936[3]).doubleValue();
ser2.f = ((java.lang.Double)alias_fs_936[4]).doubleValue();
ser2.x1.z.im = res.z.im;
ser1.x2.z.re = ind.z.re;
res.z.re= res.r;
ser2.x1.z.re = res.z.re;
cap.omega= ((2 * cap.PI) * cap.f);
ind.omega= ((2 * ind.PI) * ind.f);
cap.g.im= (cap.omega * cap.C);
ind.z.im= (ind.omega * ind.L);
cap.g.mod=Math.pow( (Math.pow(cap.g.re, 2) + Math.pow(cap.g.im, 2)), 1.0/(2));
ser1.x2.z.im = ind.z.im;
cap.g.arg=Math.asin(( cap.g.im/cap.g.mod));
cap.z.mod=( 1/cap.g.mod);
cap.z.arg=( 0-cap.g.arg);
cap.z.im= (cap.z.mod * Math.sin(cap.z.arg));
cap.z.re=Math.pow( Math.pow(cap.z.mod, 2)-Math.pow(cap.z.im, 2), 1.0/(2));
ser1.x1.z.im = cap.z.im;
ser1.x1.z.re = cap.z.re;
ser1.z.im= (ser1.x1.z.im + ser1.x2.z.im);
ser1.z.re= (ser1.x1.z.re + ser1.x2.z.re);
ser2.x2.z.im = ser1.z.im;
ser2.x2.z.re = ser1.z.re;
ser2.z.im= (ser2.x1.z.im + ser2.x2.z.im);
ser2.z.re= (ser2.x1.z.re + ser2.x2.z.re);
ser2.z.mod=Math.pow( (Math.pow(ser2.z.re, 2) + Math.pow(ser2.z.im, 2)), 1.0/(2));
```

The algorithm contains Java API method calls, however we do not show and explain the actual implementation until the next chapter.

5.6.3 Computational problem with a loop

The previous example showed how to compute circuit's impedance providing a value of frequency and distributing it over the components using the alias. In the current example, a computational problem is to find the resonance given a range of frequencies. A loop is required to solve this problem. We can use a higher-order functional dependency and add it to the extended specification of RLCseries.

```
specification Example2 extends RLCseries {
    double min, step, max, result;
    [ ser2.f -> ser2.z.mod, ser2.z.arg ], min, step, max -> result {tabulate};
    min, step, max -> result;
}
```

The main goal is to compute a `result` variable that denotes the resonance frequency. This goal however requires solving a subtask: compute `ser2.z.mod` and `ser2.z.arg` from a given value of `ser2.f`. From the previous example we know

that the specification contains enough information to solve the subtask and its algorithm is almost the same as the evaluation algorithm for the goal `f -> ser2.z.mod`, but without the distribution of the frequency. Inputs `min, step, max` denote respectively the starting frequency value, the frequency increment at each iteration and the final value. The value of `ser2.z.arg` closest to zero will signify the resonance. The realization of the higher-order function *tabulate* is given in pseudocode:

for $k = min$ **to** max **do**
  $(mod, arg) \leftarrow subtask(k)$
  $writetable(k, mod, arg)$
  $k \leftarrow k + step$
  if $abs(arg) < min_{arg}$ **then**
    $min_{arg} := abs(arg)$
    $f := k$
  end if
end for
return f

If the main computational problem is given inputs `min = 40.0`, `max = 55.0` and `step = 0.1`, after invoking the planning and executing the evaluation algorithm, the following table is produced:

ϕ	ser2.z.mod	ser2.z.arg
40.00	14.66	-1.5377
40.10	14.50	-1.5363
...		
50.20	0.53	-0.3144
50.30	**0.50**	**-0.0733**
50.40	0.51	0.1759
50.50	0.54	0.4048
...		

and the resonance frequency is `50.3`. This is the value the function *tabulate* returns and it is assigned to the variable `result`. In order to get more precise frequency value, smaller step is required.

The given example shows that it is possible to specify the problem domain using our specification language with very little effort. The concepts are implemented in the bottom-up manner starting with simplest things and developing more complex concepts using inheritance.

Chapter 6

Implementation

The implementation of the the specification language introduced in the previous chapter has been done in the CoCoViLa framework [5]. It is a Java language based software platform focused on the development of visual domain-specific languages and automatic synthesis of programs. CoCoViLa is being developed in the Software Department of the Institute of Cybernetics. The initial design and implementation was made by Ando Saabas [72]. The general programming technology was described in [23, 24]. The author of the present dissertation contributed to the project by implementing the higher-order attribute semantics, planning strategies, program extraction and a number of language and user interface improvements. In the current Chapter, these contributions will be presented and discussed.

Our goal is to provide a tool for developing domain-specific languages (DSLs). This tool will be used for specifying domain concepts (e.g. *Complex*, *Branch*, etc. in Section 5.6) and performing computations for solving given problems. A DSL developed in CoCoViLa is a specification language from Chapter 5 extended with domain-specific concepts.

6.1 Metaclasses

The first question that has to be answered is how the specification language is implemented in CoCoViLa. The specification language allows to describe concepts in terms of compound types and define the computability of variables using bindings and functional dependencies. Functional dependencies as described in Section 5.1 have realizations which are not expressible using the specification language (except

equations). In our implementation, realizations of functional dependencies including higher-order ones are Java methods. However, methods in Java, which is an object-oriented language, cannot exist without Java classes. Our design (beginning with [72]) was to wrap the specification of a concept into a Java class with the same name and include the realization of functional dependencies as methods in this class. That is, a specification of a concept becomes a meta-level specification of a corresponding Java class. We call such Java classes *metaclasses* and the corresponding specifications – *metainterfaces*. In CoCoViLa, metainterface is the central part of a concept that determines the behavior of the *object* (instance of a metaclass).

Example. A concept *Foo* with a functional dependency with a realization *doSomething* has the following metaclass:

```
public class Foo {
    /*@
        specification Foo {
            int x, y;
            ...
            x -> y {doSomething};
        }
    @*/

    public int doSomething( int value ) {
        return ...;
    }
}
```

◇

It is useful to notice that metainterfaces in metaclasses are included as comments (denoted by /*@ and @*/), so that CoCoViLa can use the information provided by metainterfaces, but the Java compiler can ignore it during the process of compilation.

Example. To make the example more concrete, we present a realization of the higher-order functional dependency from the example in Section 5.6. It is important to notice that a metainterface and a Java class have separate name spaces. Only class and method names are available in a metainterface. In the current implementation, however, one should not declare Java class members with same identifiers as metainterface variables to avoid collision of names during the code generation.

```
public class Table {
    /*@
        specification Table {
```

```
            ...
            [ ser2.f -> ser2.z.mod, ser2.z.arg ], min, step, max -> success {tabulate};
        }
    @*/

    public boolean tabulate(Subtask s, double from, double inc, double to) {
        for(double k = from; k < to; k += inc) {
            System.out.println(k + " " + s.run(k));
        }
        return true;
    }
    ...
}
```

The role of an interface *Subtask* and the corresponding method *run* will be explained later in this Chapter. ◇

6.2 Visual representation of the specification language

CoCoViLa provides a functionality for visual representation of specifications. A concept together with its visual representation specified in CoCoViLa is called a *visual class*. Visual classes are used to compose schemes in the visual environment of CoCoViLa. For each visual class the following elements are defined:

- Java class that includes a specification (a metaclass);
- visual image – a vector and/or raster graphics to be used in a scheme;
- set of ports – ports indicate which components of a class can be visually connected to other components in a scheme;
- fields – components the values of which can be given in a scheme;
- icon image – a small raster picture that is placed on a toolbar.

The collection of visual classes for a domain is called a *package*. Descriptions of packages are stored in the XML-based format described in [72]. Each package implements a domain-specific visual language (DSVL). The hierarchy of languages is shown in Figure 6.1.

For composition of schemes, three operations are defined:

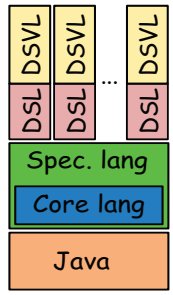
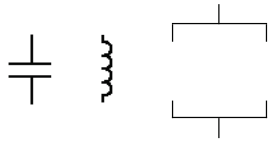


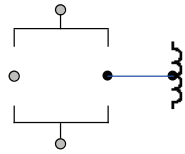
Figure 6.1: The hierarchy of languages in CoCoViLa

- Instantiating a *visual object* from a visual class. This corresponds to the declaration of a variable in the specification language.



```
Capacitor cap;
Inductor ind;
Par par;
```

- Connecting two ports. This corresponds to the binding of two variables.



```
par.x2 = ind;
```

- Providing a value to a field. This is a valuation in the specification language, i.e. an assignment of a constant to a variable (a value may or may not be shown in a scheme).

```
cap.C = 0.001;
```

Composed schemes can be saved and loaded. The format of scheme files with an extension `.syn` is also XML-based and described in [72].

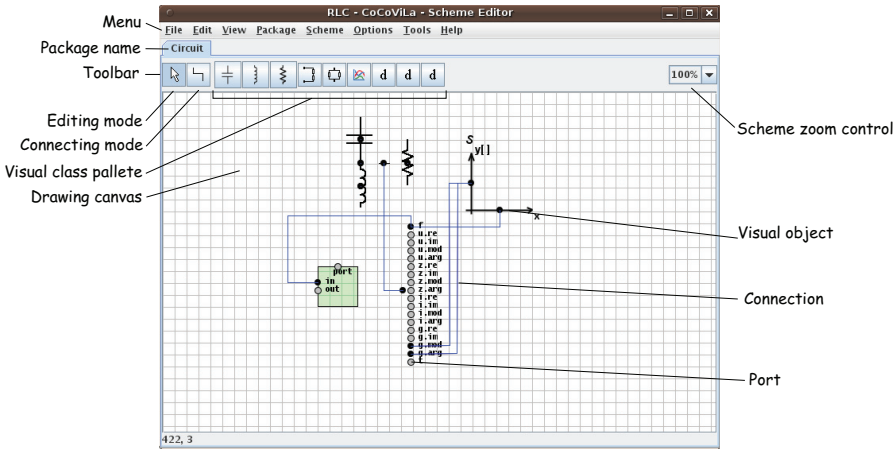


Figure 6.2: Scheme Editor window

6.3 CoCoViLa Scheme Editor

The *Scheme Editor* is a tool for visually creating schemes and executing them. To execute a scheme means to solve a computational problem specified by a scheme and a given goal, i.e. to synthesize an evaluation algorithm, to extract a program from the algorithm and invoke the program. Scheme Editor is also able to work purely on the specification level, i.e. without any schemes, statements in a flat language can be given textually.

The Figure 6.2 depicts the main window of the Scheme Editor. For each visual language (package), the workspace is automatically generated from the corresponding XML description allowing a user to compose, edit and use schemes in computations through package-specific menus and toolbars. The *File* menu allows to save, load and export schemes. The *Edit* menu contains items for undo/redo actions, cloning, searching, selecting and deleting the visual objects. The *View* menu enables to hide or show the grid, ports and names of visual objects. The *Package* menu is used for loading and closing the packages. Through the *Scheme* menu it is possible to open a specification window, a window for manual extension of a specification, a scheme options dialog, and to instantly run a composed scheme using several modes editable in this menu. The *Options* menu is for changing program's settings and the

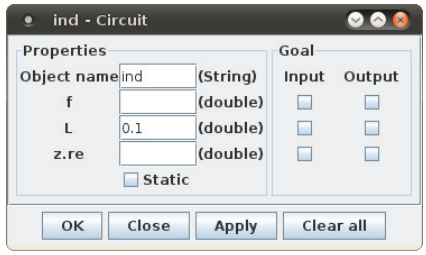
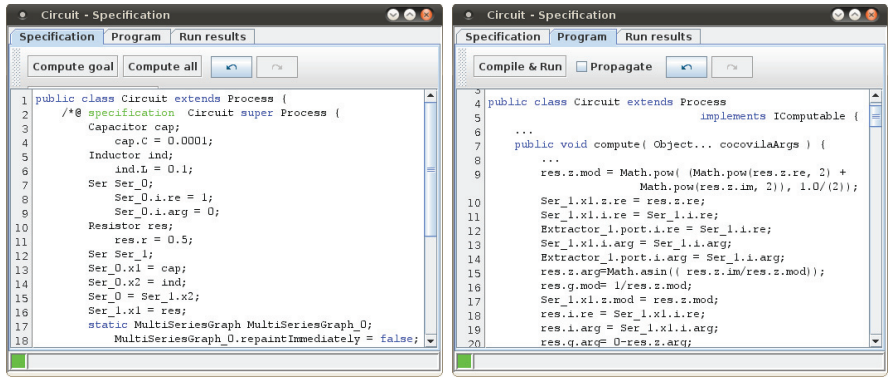


Figure 6.3: Object Property window of inductor object

way it looks. The *Tools* menu allows accessing extra features, such as an editor of decision tables, a threads dialog and an algorithm viewer.

In the Figure 6.2 the package for simulating alternating current circuits presented in Section 5.6 is loaded. *Canvas*, the area for composing schemes contains instances of a capacitor and an inductor connected serially and a resistor in parallel connection. Other components on a scheme are used for drawing charts, extracting values and running a process. For each visual object on a canvas, the pop-up menu is accessible by a right-click of a mouse allowing to manipulate an object (delete, clone, rotate, reorder, etc.) and also access the corresponding *Object Property Window*. A list of defined fields for inductor is shown in the Figure 6.3. This window also enables to specify a computational problem by defining inputs and outputs of a goal.

A scheme has a corresponding textual specification included into an automat-



(a) Specification of a scheme (b) Generated code of a program

Figure 6.4: Specification window

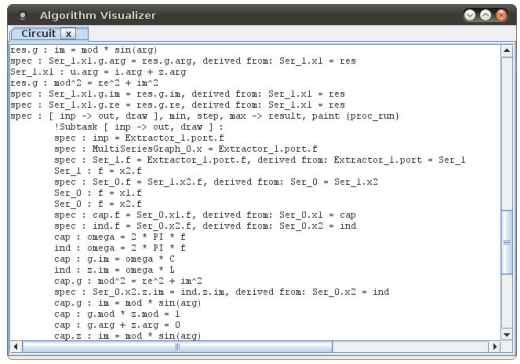


Figure 6.5: Algorithm visualizer window

ically generated metaclass. Users can open the specification window (Figure 6.4a), review and edit the specification of a scheme. Both UI buttons *Compute Goal* and *Compute All* start the planning procedure, the former button also employs the optimization procedure. The generated code, that is, the source text of a program to be executed is shown in the second tab of the window (Figure 6.4b). The code can be edited, compiled and executed at runtime. If *Propagate* is checked, values are sent back to the scheme and can be shown during the execution. The third tab *Run results* shows the list of computed values of a program and contains UI buttons for re-executing a program one or more times.

Viewing and editing specifications is not always needed. In this case users can run schemes from the menu *Scheme* → *Run*. For larger schemes it is sometimes hard to review long specifications and the generated code. For helping developers to debug their programs, *Algorithm visualizer* window can be used (Figure 6.5). It shows the pseudo-code of generated programs with appropriate indentation for nested subtasks.

The dataflow in Scheme Editor is shown in Figure 6.6 and reflects the high-level architectural design of CoCoViLa. The rectangular nodes correspond to the data, elliptic nodes represent program modules. The central part is the *Specification*. It is automatically generated from a *Scheme* or typed manually. The specification is passed to the *Parser* which constructs the corresponding *Attribute model*. The *Planner* takes attribute model as its input and produces the *Algorithm* for evalu-

ating the attributes. The *Generator* outputs the Java source code taking necessary information from an algorithm and a specification. The source code is compiled and executed. A running program can communicate with a scheme and a specification via a specialized CoCoViLa's API.

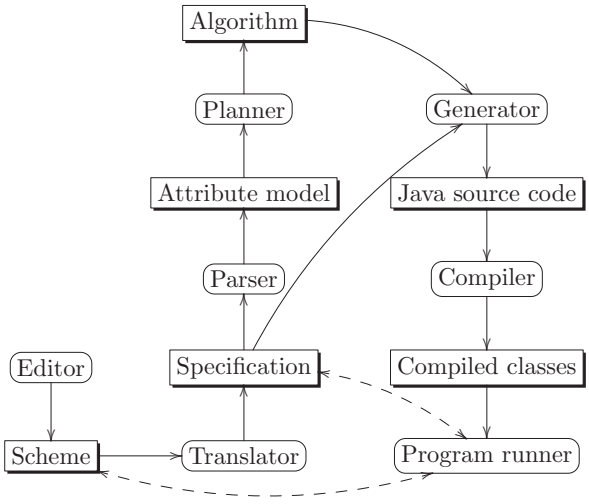


Figure 6.6: Dataflow of CoCoViLa

6.4 Additional features

A number of important features have been implemented by the author extending both the specification language and CoCoViLa's user interface. The description of some features is given in the present section.

- **Static variables.** A specification variable can be declared with `static` modifier. This will prevent the code generator to create copies of a static variable in the subtask environments.

- **Constants.** Constant values cannot be overridden during computations. For example: `const double PI = Math.PI;`. In Java code, a constant will be a class member with *final static* modifiers.

- **Independent subtasks.** This feature changes the logic needed for representing the semantics of specifications (see [51]), but it can be easily explained in terms of attribute models. It has been implemented also in earlier SSP systems. Subtasks in higher-order attribute dependencies are called *dependent* because solving a subtask may require to utilize attributes from an outer context. Independent subtasks change the notion of subtasks in the following way. Besides defining input and output attributes of a subtask, a third argument should also be given, that is, the name of a context (i.e. a compound type) on whose attribute model a subtask should be solved.

  $$IndependentSubtask ::= [Context \vdash VarList \ \rightarrow \ VarList]$$

  The synthesized algorithm of an independent subtask is cached and can be used in different specifications without re-invoking the planning procedure.

- **Calling planner from Java.** Independent subtasks allow defining computational contexts, this could also be useful for tasks, where specifications are constructed at runtime. In this case, the planner can be called from the Java code using a special method `computeModel()` in class `ProgramContext` of CoCoViLa API. The method has the following signature:

  ```
  Object[] computeModel(String context, String[] inputNames,
                        String[] outputNames, Object[] inputValues)
  ```

  The evaluation algorithm of a subtask is planned and executed at runtime and the method returns computed values to the calling program.

- **Multiports.** A connection to an ordinary port in a scheme constitutes a binding between two variables. If a port represents an alias, it can only be connected to another port with an alias of the same structure. In some cases, the structure of an alias may not be known and should be defined in a scheme. Another type of ports is required. *Multiport* is such a ports that represents

an alias that is declared but not initialized in the specification of an object. Each connection to such port does not create a binding, a connected variable becomes an element of the alias behind the multiport. A multiport with no connections corresponds to an empty alias. Connections between two multiports are not allowed.

- **Scheme superclass.** In certain cases schemes in a package need additional functionality for manipulating objects on a canvas. For instance, an alias with a wildcard has to be defined for binding variables of the objects. This alias could also be an input or output of a functional dependency. Such a specification and realizations of functional dependencies can be written in a metaclass. From the CoCoViLa's UI, a scheme can inherit this metaclass. In this case, a wildcard will work with objects defined in a scheme. Multiple schemes can inherit the same superclass to reuse its specification and methods.

- **Scheme specification extension.** Specifications of schemes can also be extended without using superclasses. An extension of a specification is written in a special window and saved in XML file of a scheme. Such extensions are scheme-specific and cannot be shared between the schemes. This feature is well suitable for fast debugging and testing of a scheme by providing values of variables of objects in a scheme and specifying equations, bindings and goals. Functional dependencies can only be written for methods implemented in a superclass of a scheme (if there has been defined one).

6.5 Planning strategies

Computational problems from different domains have different natures. Some problems can be solved without the use of subtasks, some problems require nested and/or repetitive subtasks, others may need subtasks executed sequentially. In other words, there is no general strategy to solve any problem in an optimal way. Thus, different heuristics can be used for tweaking the planner.

6.5.1 Depth-first search

By default, the planning algorithm works almost as described in Chapter 4. It is the depth-first search on the and-or tree of subtasks. The difference is that it has a restriction on branches of subtasks. There can be at most one occurrence of a particular subtask in one branch, i.e. the repetition of nested subtasks is not allowed.

6.5.2 Incremental depth-first search

Incremental depth-first search algorithm employs the depth limiting strategy. The algorithm starts with the depth limit equal to one and increments the bound at each iteration. It allows to find the shallowest goal state faster than the ordinary depth-first search. Also, the algorithm prunes unnecessary branches, i.e. on each iteration it does not visit all states from the previous depth.

6.5.3 Allowing subtask repetitions

Some problems require the repetition of subtasks for solving a computational problem, that is, when subtasks need to be solved using same subtasks in the same branch. This setting requires limiting the depth of a search tree, otherwise the planning may not terminate in case of a depth-first search. Choosing a correct depth in some cases is crucial for the search time. When the maximum depth of a search tree that is sufficient for solving a problem is unknown, it is suggested to use incremental depth-first search planning algorithm.

Chapter 7

Applications

This Chapter describes real-world applications that have been developed by experts in various domains with participation of the author of the present thesis. The contribution of the author has been in participation of design of applications, consulting and also adaptation of CoCoViLa and introduction of some new features to suite the needs of new domains, e.g. independent subtasks, extension of alias construct, multiports, decision tables, etc. These examples should demonstrate the applicability of the results obtained in this thesis for real-life applications in different problem domains.

7.1 Composition of web services: X-Road

In the context of this work, first, the automatic web service composition methodology was proposed and, second, this methodology was applied for developing an experimental software package in CoCoViLa for the maintenance and extension of an existing Estonian e-government information system called the X-Road. Several papers have been published covering various aspects of this methodology and application [44, 45, 46, 50].

X-Road guarantees the secure access to most of the national databases of Estonia by means of web services through domain-specific portals available to the residents having national identification number. X-Road allows the usage not only of atomic services but also a compound ones (i.e. composed chains of services). CoCoViLa provides good infrastructure for service developers to perform automatic composition with the very little effort. The syntactic model of the operational part of X-Road has been created and transformed into the format of CoCoViLa. The model is represented visually on a scheme and includes about 300 atomic services

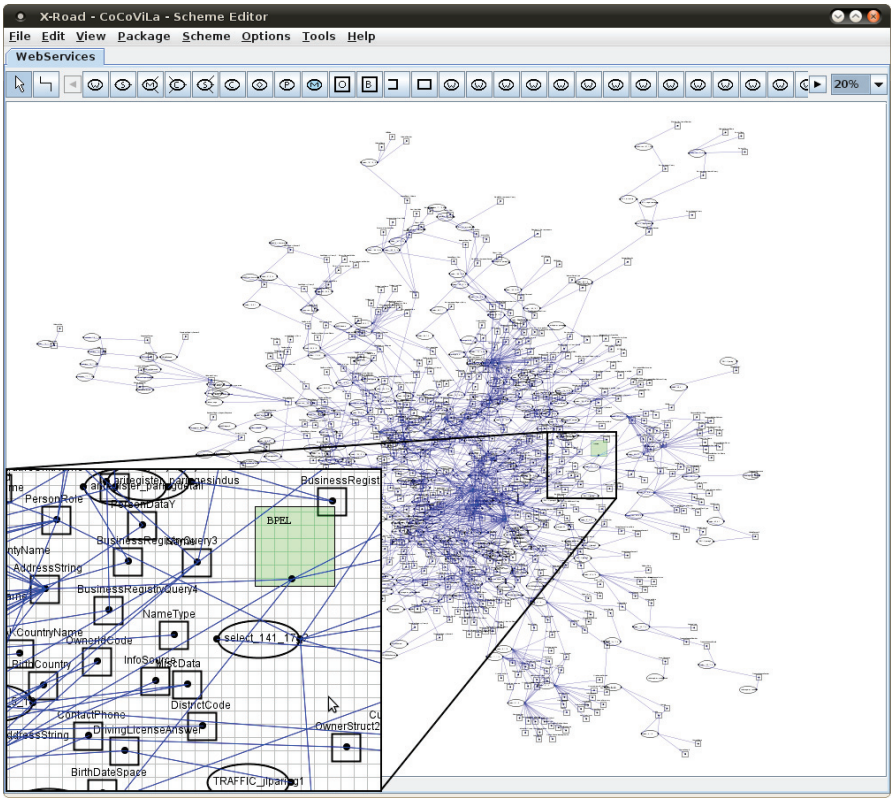


Figure 7.1: X-Road model in CoCoViLa

(with corresponding metaclasses) and about 600 connections to semantic resources, see Figure 7.1. In the figure, services are represented by ovals and data resources by squares. Users can provide input data to initial services on the scheme and state a goal for finding a chain of services to compose a complex service. CoCoViLa automatically synthesizes a program that generates a BPEL or OWL-S description of a desired complex service. The program is synthesized only if the final services are composable from the initial ones and all required data are made available. This guarantees the correctness of composed services. This application shows that CoCoViLa provides good visual support for model maintenance and is able to handle large-scale models while keeping synthesis time at a significantly low level.

7.2 Simulation of hydraulic systems

Modeling and simulation of hydraulic-mechanical systems (e.g automatically regulated fluid power systems of stationary and mobile machines, steering mechanisms of cars and ships, drives of robots, etc.) has been investigated in Tallinn University of Technology for several decades. Before CoCoViLa, older systems were used (NUT, Priz, etc) that had considerable limitations on memory usage and the expressiveness of the specification language. CoCoViLa has enabled the researchers to take the modeling and simulations of hydraulic-mechanical systems to a much higher level, providing improved usage of aliases (no restrictions on the nestedness), independent subtasks, "out of the box" memory management due to Java VM and better visualization capabilities.

Hydraulic elements (hydraulic motor, pump, resistors, volume elasticities, tubes, interface elements, etc.) from older packages [25] for modeling and simulating hydraulic systems have been implemented in CoCoViLa [26]. Some of the elements are visible in the scrollable toolbar in Figure 7.2. The developed package includes also visual classes for drawing charts and a simulation engine based on iterative methods (Runge-Kutta) for solving ordinary differential equations. This package enables one to hierarchically construct mathematical models of large and complicated hydraulic systems that include thousands of variables, equations and functions. The package is used to simulate steady state conditions and the dynamic behavior of the hydraulic load-sensing systems.

Figure 7.2 shows an example of a simulation of steady-state conditions. The Scheme Editor contains a scheme that represents the multi-pole model of a hydraulic-mechanical load-sensing system. The second window contains the corresponding textual specification of the model. The unfolded model of this problem includes 1988 variables and 4532 attribute dependencies. The generated simulation program extracted from the attribute evaluation algorithm is a Java class that has 4958 lines of Java code. Attribute evaluation planning and code generation for this example takes about half a seconds on a typical 2.0 GHz laptop. The window in the foreground contains the result of this simulation – a 3D chart with calculated 1000x1000 points. It shows the efficiency coefficient of the load sensing system depending on the displacement of the directional valve and the load moment of the hydraulic motor.

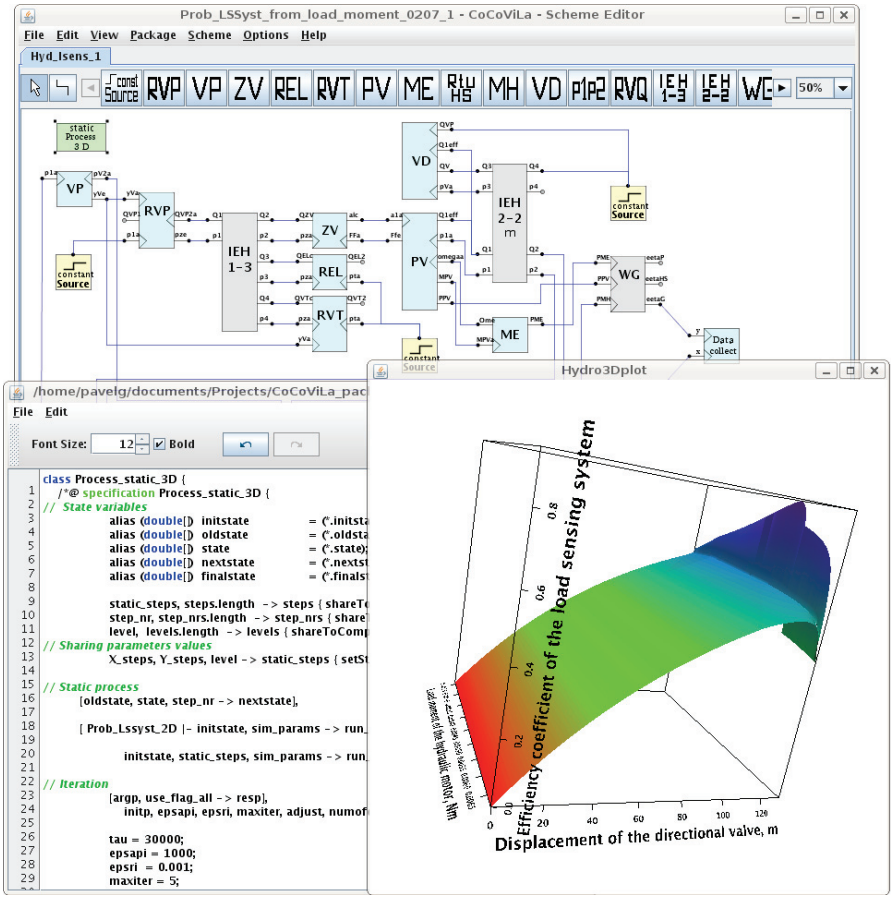


Figure 7.2: Simulation of a hydraulic-mechanical load-sensing system

7.3 Simulations in cyber-security

The work on appying response measures against cyber attacks rises from the fact that the number of computer systems connected to the Internet and infected with malware is increasing, as a consequence, leading to the higher probability of large-scale denial-of-service attacks. Graph-based Automated Denial-of-Service Attack Response (GrADAR) [33] is an approach where the selection of responses is made according to an estimation of an impact of particular counter-attack measures. Co-

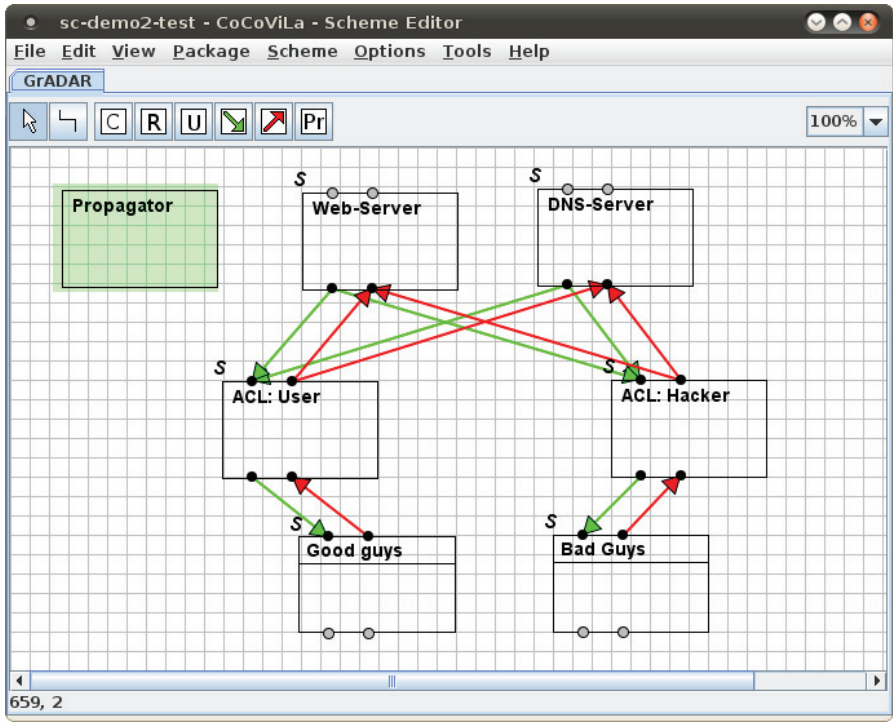


Figure 7.3: Visual specification of a response analysis problem in GrADAR package

CoViLa was used to create a GrADAR software package for visual modeling of graphs representing networks containing information such as dependencies between resources and their availability and to implement algorithms for automatic selection and application of responce measures. This is a joint work between the following organizations – Fraunhofer Institute for Communication, Information Processing and Ergonomics FKIE (Germany), Cooperative Cyber Defence Centre of Excellence and Institute of Cybernetics (Tallinn, Estonia) [37].

Figure 7.3 shows a scheme in CoCoViLa representing a network of resources with connections for propagating workload and availability values (red and green arrows). The goal is to analyze the effect of response measures. CoCoViLa allows not only to enter the parameter values of components using the graphical user interface, but also due to absence of restrictions on the usage of Java language and libraries, it allows

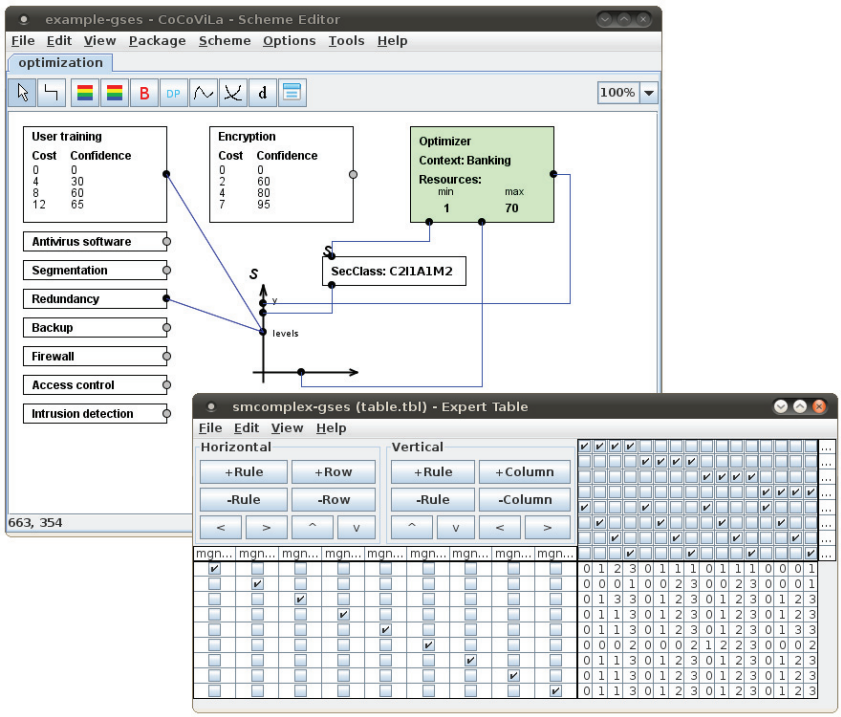


Figure 7.4: Visual specification and expert table editor

integration with other software, e.g. back-end managements systems to receive live values into the running program synthesized from the scheme.

7.4 Graded security expert system

This section concerns a work [36] in the field of modeling of graded security measures which are used e.g. in banking and energy sectors. The graded security model is intended to help determine reasonable or optimal sets of security measures according to the given security requirements. An expert system with visual specification language for security system description shown in Figure 7.4 has been implemented as a software package in CoCoViLa. The system together with visual components and optimization algorithm for calculation of Pareto curves, includes a set of rules

(knowledge modules) in the form of decision tables that help handling the expert knowledge. Expert tables have been implemented in the context of this application but designed to be functional in other problem domains that require handling of the expert knowledge.

7.5 Functional constraint networks

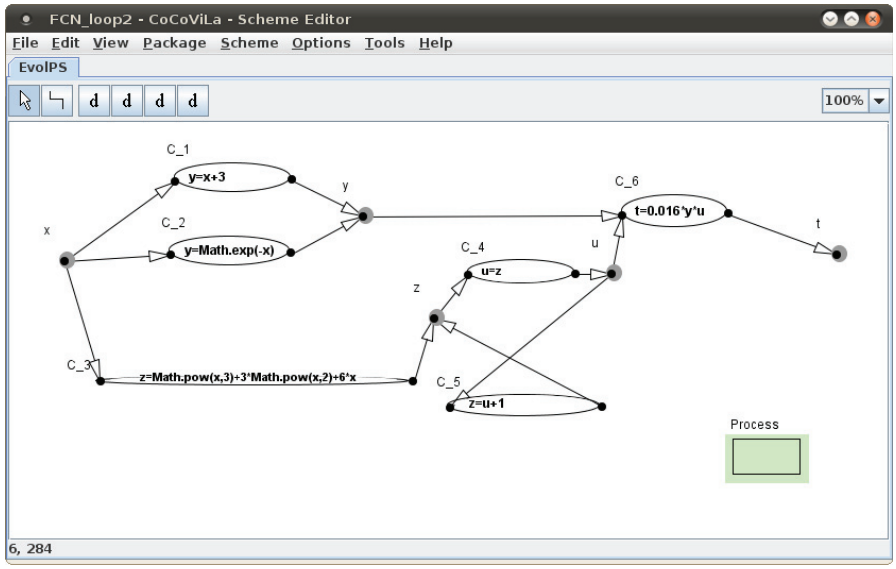


Figure 7.5: Visual specification of FCN

In the context of the work by J. Sanko and J. Penjam on inductive approach for automatic program construction from formal specifications represented by functional constraint networks (FCNs) [73], a package prototype in CoCoViLa has been implemented. Using the package, first, a functional constraint network is visually specified in a scheme (Figure 7.5) that describes an input-output behavior of a program to be synthesized. Second, a state transition machine (STM) that represents all possible solutions (if any exist) to the specified computational problem, is automatically derived from a given FCN. Thereafter, a search algorithm that employs evolutionary computation technique, in particular, differential evolution, is recursively applied on

a STM to produce a set of programs that satisfy the specified input-output behavior. Each program generated by traversing a STM is transformed into JavaScript code and verified at runtime using SUN Java's built-it JavaScript interpreter. As a result of the computation, one or more programs closer corresponding to the given specification are returned.

7.6 Educational packages

It should be also mentioned that a number of analytical and simulation packages for neural networks, electrical and logical circuits, mechanical drives, attack-trees, etc. have been implemented in CoCoViLa and used for educational purposes in several undergraduate and graduate courses at Tallinn University of Technology.

Chapter 8

Planner's benchmarking

This Chapter provides an evaluation of CoCoViLa planner's performance and analysis of its efficiency for program construction. The experiment is performed in comparison with several logic theorem provers.

8.1 Problem statement

It has been noted in Section 4.5 that our planning of an attribute evaluation algorithm logically corresponds to a derivation of an intuitionistic propositional formula. Therefore, it is reasonable to compare the planner with other intuitionistic logic provers. Looking at the planning algorithm we see that the complexity of search increases significantly when subtasks have to be solved in a nested way, and especially when one and the same subtask has to be used repeatedly in one and the same branch. We are going to formulate increasingly complex tasks to solve for CoCoViLa, and respective (logically equivalent) theorems for other provers and compare the proof times. It should also be mentioned that program synthesizers have already been used for proving logical theorems [29, 91].

The Kripke's formula $((((A \to B) \to A) \to A) \to B) \to B$ used in Section 4.5 can be extended in such a was that its proof will include two applications of one and the same subtask in a branch. This will be used as a building block for developing more complex tasks.

The original formula contains nested implications and has to be encoded in a suitable format to reduce the depth of nestedness of implications to one (to precisely match the form of attribute dependencies in attribute models). Such encoding has

been used in structural synthesis of programs [51, 60]. Tammet [80] applied the similar procedure called *labelling* to reduce the depth of a formula.

For any propositional formula G containing subformula F, the encoding is done using the equivalence replacement theorem $(X \leftrightarrow F) \rightarrow (G \leftrightarrow G_F[X])$, where $G_F[X]$ is the formula with variable X substituted for F in G. After replacing a subformula with a variable, new formulas have to be added into the sequent to preserve the derivability of initial formula. The present example deals only with implicational fragment, thus only the replacement of a subformula $(U \rightarrow V)$ is considered. To replace $(U \rightarrow V)$ with X, the following formulas should be added: $X \rightarrow (U \rightarrow V)$, $(U \rightarrow V) \rightarrow X$. The first formula is equivalent to $X\&U \rightarrow V$. Note that in SSP such formulas are called *axioms*.

The Kripke's formula is encoded as follows:

- Formula is represented in the sequent notation to explicitly show a goal B:

$$((((A \rightarrow B) \rightarrow A) \rightarrow A) \rightarrow B) \vdash B.$$

- Subformula $(A \rightarrow B)$ is replaced by X and new formulas are added:

$$\begin{aligned} (A \rightarrow B) &\rightarrow X, \\ X\&A &\rightarrow B, \\ ((X \rightarrow A) \rightarrow A) \rightarrow B \quad &\vdash \quad B \end{aligned}$$

- Subformula $(X \rightarrow A)$ is replaced by Y:

$$\begin{aligned} (X \rightarrow A) &\rightarrow Y, \\ Y\&X &\rightarrow A, \\ (A \rightarrow B) &\rightarrow X, \\ X\&A &\rightarrow B, \\ (Y \rightarrow A) \rightarrow B \quad &\vdash \quad B \end{aligned}$$

The proof of the theorem using the rules of structural synthesis can be found in [51]. The proof shows that formulas $(X \rightarrow A) \rightarrow Y$ and $X\&A \rightarrow B$ are not needed to derive a goal B. The sequent is developed further:

- Formulas $(X \to A) \to Y$ and $X\&A \to B$ are removed.
- A new variable G is introduced in the following formulas:

$$(Y\&G \to A) \to B,$$
$$(A \to B\&G) \to X$$

  This ensures that the second formula has to be used only after the first in the derivation tree to keep the complexity of proof at a higher level.

- A new variable U is added into the following formula:

$$Y\&X\&U \to A$$

- A new variable Z is added into the formula:

$$(A \to (B\&G\&Z)) \to X$$

- Two assertions U and Z are added to the sequent. The necessity of variables U and Z will be explained later.

The final sequent has the following form:

$$\begin{aligned} (Y\&G \to A) &\to B, \\ (A \to (B\&G\&Z)) &\to X, \\ Y\&X\&U &\to A, \\ U, & \\ Z \quad &\vdash \quad B \end{aligned}$$

The derivation of the sequent using intuitionistic system GJ' formulated by Mints [58] is presented in Figure 8.1.

The idea behind this set of formulas was to construct a theorem in such a way that it would be possible to make a copy of the set of formulas (by renaming the variables) and include a copied set into existing sequent. Variables G, U and Z help to achieve this, their equivalences will bind copies of the set of formulas and they are all needed to guarantee the deep nestedness of proofs of subtasks by forcing an

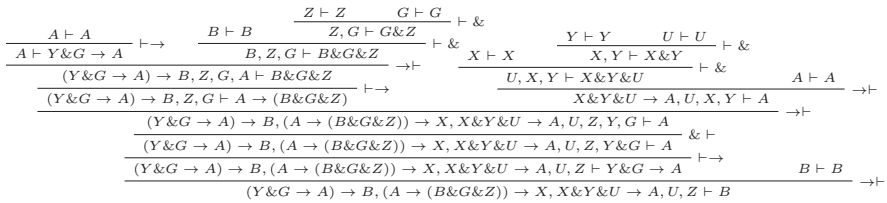


Figure 8.1: Proof of extended formula in intuitionistic sequent calculus GJ′

order on the application of the derivation rules (to preserve the highest complexity of a proof). For two sets (copies) of formulas, variable Z_1 of the first set is bound by equivalence with variable B_2 of the second set and Y_1 is bound with U_2. The n-level sequent has the following form:

$$
\begin{aligned}
(Y_1 \& G_1 \to A_1) &\to B_1, \\
(A_1 \to (B_1 \& G_1 \& Z_1)) &\to X_1, \\
Y_1 \& X_1 \& U_1 &\to A_1, \\
Z_1 &\leftrightarrow B_2, \\
Y_1 &\leftrightarrow U_2, \\
&\dots \\
(Y_n \& G_n \to A_n) &\to B_n, \\
(A_n \to (B_n \& G_n \& Z_n)) &\to X_n, \\
Y_n \& X_n \& U_n &\to A_n, \\
Z_{n-1} &\leftrightarrow B_n, \\
Y_{n-1} &\leftrightarrow U_n, \\
U_1, & \\
Z_n \quad &\vdash \quad B_1, \\
\text{where } n &\geq 2
\end{aligned}
$$

For example, in CoCoViLa, the specification of a second-level sequent is as follows:

```
/*@ specification Kripke2
```

```
    boolean A1,B1,X1,Y1,Z1,G1,U1;
    [Y1, G1 -> A1] -> B1 {f1};
    [A1 -> B1, Z1, G1] -> X1 {f2};
    X1, Y1, U1 -> A1 {f3};
    boolean A2,B2,X2,Y2,Z2,G2,U2;
    [Y2, G2 -> A2] -> B2 {f1};
    [A2 -> B2, Z2, G2] -> X2 {f2};
    X2, Y2, U2 -> A2 {f3};
    Z1 = B2;
    Y1 = U2;
    U1, Z2 -> B1;
@*/
```

Due to the fact that CoCoViLa is not a theorem prover, but a program synthesizer, this specification contains some redundant information (i.e. variable declarations and function names `f#` in realizations of attribute dependencies) not relevant to the problem under consideration. This information should be ignored by the reader because the generated code, extracted from the proof is not executed (there are no methods `f#()` in the corresponding Java class). However, one could assign some computational meaning to this specification. A computational meaning of initial Kripke's formula and realizations of formulas in the specification have been demonstrated by Lämmermann [39].

The complexity of proof search in intuitionistic logic is PSPACE-complete [78]. We expected that the time of proof search would grow exponentially with the growth of the nestedness of subtasks in a proof.

The goal of this experiment is to check and compare the performance of CoCoViLa planner to various intuitionistic logic automated theorem provers publicly available for download on the Internet. The following theorem provers were used in the benchmark (referred on ILTP library homepage [32]): STRIP [41], iLeanCoP [66], iLeanSeP [31], iLeanTAP [65], LJT [19], Gandalf [80], PITP [2].

8.2 Measurements

The experiment was conducted on a 2.0 GHz laptop with 2 GB of RAM and a desktop version of Ubuntu 10.04 operating system. The decision was made to set the time bound for the proof search to one and a half hours.

8.3 Results

The results of the benchmark are summarized in the Table 8.1. Time is shown in seconds. Dashes mean that a prover was not able to solve a given problem within the time bound.

- **CoCoViLa.** The planner was tested with two strategies, depth-first search with and without incremental deepening (DFS and IDFS correspondingly). The initial results were not very promosing due to the design of problem specification. CoCoViLa is a visual framework and the first attempt was to represent a single level of Kripke's sequent as a separate component with visual representation. The n-level sequent was constructed visually in a scheme binding variables using aliases. This approach created unnecessary overhead and the decision was made to write specifications textually the same way it was done for the rest of contestants. First, the IDFS strategy was tested. The results are presented in the first row. In CoCoViLa, proof trees for n-level sequents had the maximum depth of $n+2$. Because of the fact that the solution to Kripke's sequent requires the repetitive usage of subtasks in same branches, the second strategy (DFS) needed the restriction on the depth of search trees. The depth limit was set to $n+2$ for each n-level sequent.

- **STRIP.** This is a C implementation of decision procedure based on contraction-free sequent calculus for propositional formulas. The rule-prec stategy which was the default configuration was used for the proof search. To a provable formula STRIP generated its proof tree, but it was also able only to check the derivability of a formula. In case of unprovable formula, it produced a Kripke countermodel. Our test showed that checking derivability was much more efficient than generating a proof. In fact, STRIP was able to generate a proof tree only up to 2-level Kripke's sequent within the time bound. The derivability check succeeded in proving 6-level sequent which took a bit more than one hour.

- **iLeanCoP.** This is an automated theorem prover for intuitionistic first-order logic based on the clausal connection calculus for intuitionistic logic imple-

mented in Prolog. Version 1.2 was used in the benchmark running under ECLiPSe Prolog system version 5.10. The prover does not generate proof trees. Suprisingly, iLeanCoP was able to handle only the first-level sequent.

- **iLeanSeP.** This is another automated theorem prover from the Lean family. It is based on a single-succedent intuitionistic sequent calculus. SWI-Prolog version 5.7.8 was used for running the prover. This prover also does not output proof trees. The prover showed similar results to iLeanCoP being able to prove only the first-level sequent.

- **iLeanTAP.** It is the first-order theorem prover based on semantic tableaux method. For some reason it did not perform well even with the first-level sequent.

- **LJT.** Roy Dyckhoff's intuitionistic theorem prover based on the Vorobjev-Hudelmaier calculus [19] implemented in Prolog. Without possibility to generate proof trees, it demonstrated good performance being able to decide the provability of 6-level sequent in less than a minute and also prove the 7-level sequent within the time bound.

- **Gnonclassic/GInt.** Tanel Tammet's intuitionistic version of award-winning Gandalf system is a predicate-level theorem prover based on resolution calculi due to Maslov and Mints. It is implemented in Scheme and compiled into C for efficiency. Gandalf demonstrated an exceptional performance showing the best timings, especially for high-level sequents. The time measured included program start-up, formula parsing and preparations. For provable formulas, Gandalf always generates and outputs proof trees.

- **PITP.** This is a tableau prover for intuitionistic propositional logic with $O(n\log n)$-SPACE decision procedure implemented in C++. Version 3.1 was used in the test. The prover only checks the provability of formulas without generating proof trees. The timing in the table was taken from prover's statistics and stated to be clean proof search time. Preparations took some additional time not reflected in the table. It also appeared that the prover's efficiency highly depended on the order of formulas. We were able to generate

	Level of extended Kripke formula							
Tool	1	2	3	4	5	6	7	10
CoCoViLa (IDFS)	<0.01	<0.01	0.02	0.55	18.82	793.6	–	–
CoCoViLa (DFS)	<0.01	<0.01	<0.01	0.05	1.18	36.24	1579.6	–
STRIP (check)	<0.01	<0.01	<0.01	0.34	28.75	3781.3	–	–
STRIP (prove)	<0.01	34.87	–	–	–	–	–	–
iLeanCoP	0.01	–	–	–	–	–	–	–
iLeanSeP	0.02	–	–	–	–	–	–	–
iLeanTAP	–	–	–	–	–	–	–	–
LJT	<0.01	<0.01	0.01	0.05	1.04	35.15	1572.28	–
Gandalf	0.01	0.02	0.08	0.19	0.36	0.53	0.88	7.550
PITP	<0.01	0.01	0.01	0.05	0.68	15.73	343.5	–

Table 8.1: Performance evaluation of various theorem provers

a 300-level sequent with a particular order of formulas and pure search time was about ten seconds. However, this result was not included in the table.

8.4 Analysis

From the results in Table 8.1 it is clear that the problem had exponential complexity. CoCoViLa showed similar performance to the most of the provers on lower level sequents and average performance (which was expected) in comparison to the highly-optimized theorem provers such as Gandalf and PITP. However, only two theorem provers (besides CoCoViLa) were generating proof trees. In other words, the big disadvantage of provers not generating proof trees is that they cannot be used for program construction.

The real-world programs that we have dealt with so far contained the nestedness of subtasks at most of depth three and such problems were solved by the planner in a matter of milliseconds. Our experience has shown that the higher subtask dependency leads to very complex and incomprehensible specifications that are hard to design, implement and especially debug.

Chapter 9

Conclusions

The main research interest of this thesis was focused on the software development methodology for constructing programs from specifications in efficient and convenient way.

We introduced the flat languages as a class of declarative languages for specifying simple concepts as well as larger systems. From such specifications, given a problem statement, a program can be obtained that computes a required goal. The advantage of flat languages is on the one hand their structural simplicity, and on the other hand their expressive power that is enough to support hierarchy, inheritance, polymorphism and also control structures. Many domain-specific languages are in essence flat languages.

The semantics of specifications in flat languages was given by means of higher-order attribute models. It included three steps that are completely automated:

1. translation of a specification into an attribute model;

2. attribute evaluation planning after having defined a computational problem with a set of input and target attributes;

3. extraction and execution of a program for computing the attributes.

To describe higher-order attribute evaluation planning algorithm, a concept of *maximal linear branches* was used. An *optimization* algorithm was given for synthesizing more efficient programs.

We presented an instance of a flat language – the specification language that consists of the core language translatable into attribute models and extensions that

include tuples, wildcards and equations. The specification language is built on top of Java language. This enables us to use primitive and object types of Java in the specifications and specifications themselves are embedded into Java classes. This gives natural and flexible interoperability between models and the actual code. By means of higher-order functional dependencies (that correspond to higher-order attribute dependencies once translated into attribute models) it is possible to synthesize programs with loops, recursion and conditionals.

A considerable part of this work was focused on the implementation. As a result, the CoCoViLa system gained some new functionality and is able to handle large applications.

The algorithm of attribute evaluation on higher-order attribute models can be explained also in terms of logic, and vice versa – theorems of intuitionistic propositional calculus can be encoded in the form of higher-order attribute models. Hence it was interesting to compare the algorithm implemented in CoCoViLa with well-known theorem provers. This was accomplished the thesis, and as the results have shown, for a given task only two of the provers performed considerably better than the CoCoViLa's algorithms.

Bibliography

[1] Ben Abbott, Ted Bapty, Csaba Biegl, Gabor Karsai, and Janos Sztipanovits. Model-based software synthesis. *IEEE Software*, 10(3):42–52, 1993.

[2] Alessandro Avellone, Guido Fiorino, and Ugo Moscato. Optimization techniques for propositional intuitionistic logic and their implementation. *Theor. Comput. Sci.*, 409(1):41–58, 2008.

[3] Janis Barzdins and Dines Bjørner, editors. *Baltic Computer Science, Selected Papers*, volume 502 of *Lecture Notes in Computer Science*. Springer, 1991.

[4] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Trans. Program. Lang. Syst.*, 7(1):113–136, 1985.

[5] CoCoViLa homepage. http://www.cs.ioc.ee/cocovila/.

[6] Robert L. Constable. Programs and types. In *FOCS*, pages 118–128. IEEE, 1980.

[7] Daniel E. Cooke, Matt Barry, Michael Lowry, and Cordell Green. NASA's Exploration Agenda and Capability Engineering. 2006.

[8] Gennaro Costagliola, Genoveffa Tortora, Sergio Orefice, and Andrea De Lucia. Automatic generation of visual programming environments. *Computer*, 28(3):56–66, 1995.

[9] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[10] James B. Dabney and Thomas L. Harman. *Mastering SIMULINK*. Prentice Hall, 2001.

[11] Juan de Lara and Hans Vangheluwe. Atom^3: A tool for multi-formalism and meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2002.

[12] Juan de Lara and Hans Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *J. Vis. Lang. Comput.*, 15(3-4):309–330, 2004.

[13] Juan de Lara Jaramillo, Hans Vangheluwe, and Manuel Alfonseca Moreno. Using meta-modelling and graph grammars to create modelling environments. *Electr. Notes Theor. Comput. Sci.*, 72(3), 2003.

[14] Dionisio de Niz. Diagram and Language for Model-Based Software Engineering of Embedded Systems: UML and AADL. *Software Engineering Institute white paper, Carnegie Mellon University*, 2007.

[15] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute grammars: definitions, systems and bibliography.* Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[16] Pierre Deransart and Jan Maluszynski. Relating logic programs and attribute grammars. *J. Log. Program.*, 2(2):119–155, 1985.

[17] Didier Parigot, Gilles Roussel, Etienne Duris, and Martin Jourdan. Attribute Grammars: a Declarative Functional Language. Technical report, 1995. Projet CHARME.

[18] Aleksandr Dikovsky. Solution in linear time of algorithmic problems connected with synthesis of nonlooping programs. *System Programming and Computer Software*, 3:38–49, 1985.

[19] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 57(3):795–807, 1992.

[20] Gregor Engels, Christian Soltenborn, and Heike Wehrheim. Analysis of uml activities using dynamic meta modeling. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2007.

[21] Robert Esser and Jörn Janneck. A framework for defining domain-specific visual languages. In *Workshop on Domain Specific Visual Languages, in conjunction with ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Oopsla*, 2001.

[22] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–130, 1992.

[23] Pavel Grigorenko, Ando Saabas, and Enn Tyugu. Cocovila - compiler-compiler for visual languages. *Electr. Notes Theor. Comput. Sci.*, 141(4):137–142, 2005.

[24] Pavel Grigorenko, Ando Saabas, and Enn Tyugu. Visual tool for generative programming. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 249–252, New York, NY, USA, 2005. ACM Press.

[25] Gunnar Grossschmidt and Mait Harf. Multi-pole modelling and simulation of a four way valve controlled fluid power system in NUT programming environment. In Krzysztof Amborski and Hermann Meuth, editors, *ESM*, pages 677–681. SCS Europe, 2002.

[26] Gunnar Grossschmidt and Mait Harf. Modelling and simulation of a hydraulic load-sensing system in the CoCoViLa environment. In *Proceedings of the 6th International Conference of DAAAM Baltic Industrial Engineering 24-26th April 2008, Tallinn, Estonia*, pages 83–88. Tallinn University of Technology, 2008.

[27] Yuri Gurevich. The abstract state machine paradigm: What is in and what is out. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Er-*

shov Memorial Conference, volume 2244 of *Lecture Notes in Computer Science*, page 24. Springer, 2001.

[28] Görel Hedin and Eva Magnusson. Jastadd–an aspect-oriented compiler construction system. In *Science of Computer Programming*, pages 37–58, 2003.

[29] Christian Horn and Alan Smaill. Theorem proving and program synthesis with oyster. pages 425–436, 1992.

[30] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[31] iLeanSeP homepage. http://www.leancop.de/ileansep.

[32] ILTP Library homepage. http://www.cs.uni-potsdam.de/ti/iltp.

[33] Marko Jahnke, Gabriel Klein, Jens Tölle, and Peter Martini. Protecting military networks with gradar - an approach for graph-based automated denial-of-service attack response. In *Proceedings of the International Military Communication Conference (MCC 2009)*, Prague, Czech Republic, September 2009.

[34] Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, implementation and evaluation of the fnc-2 attribute grammar system. In *PLDI*, pages 209–222, 1990.

[35] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008.

[36] Jüri Kivimaa, Andres Ojamaa, and Enn Tyugu. Graded security expert system. In Roberto Setola and Stefan Geretshuber, editors, *CRITIS*, volume 5508 of *Lecture Notes in Computer Science*, pages 279–286. Springer, 2008.

[37] Gabriel Klein, Andres Ojamaa, Pavel Grigorenko, Marko Jahnke, and Enn Tyugu. Enhancing response selection in impact estimation approaches. In Marek Amanowicz, editor, *Concepts and Implementations for Innovative Military Communications and Information Technologies*. Military University of Technology, Warsaw, 2010.

[38] Donald Knuth. Semantics of context-free grammars. *Mathematical Systems Theory*, 2:127–145, 1968.

[39] Sven Lämmermann. Automated composition of java software. Technical report, Lic. thesis, Department of Teleinformatics, Royal Institute of Technology, Sweden, 2000.

[40] Sven Lämmermann. *Runtime Service Composition via Logic-Based Program Synthesis*. PhD thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, June 2002.

[41] Dominique Larchey-Wendling, Dominique Méry, and Didier Galmiche. Strip: Structural sharing for efficient proof-search. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 696–700. Springer, 2001.

[42] Michael R. Lowry, Andrew Philpot, Thomas Pressburger, and Ian Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *ISMIS '94: Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems*, pages 326–335, London, UK, 1994. Springer-Verlag.

[43] George Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 5th edition, 2005.

[44] Riina Maigre, Pavel Grigorenko, Peep Küngas, and Enn Tyugu. Stratified composition of web services. In Maria Virvou and Taichi Nakamura, editors, *JCKBSE*, volume 180 of *Frontiers in Artificial Intelligence and Applications*, pages 49–58. IOS Press, 2008.

[45] Riina Maigre, Peep Küngas, Mihhail Matskin, and Enn Tyugu. Handling large web services models in a federated governmental information system. In Abdelhamid Mellouk, Jun Bi, Guadalupe Ortiz, Dickson K. W. Chiu, and Manuela Popescu, editors, *ICIW*, pages 626–631. IEEE Computer Society, 2008.

[46] Riina Maigre, Peep Küngas, Mihhail Matskin, and Enn Tyugu. Dynamic service synthesis on a large service models of a federated governmental information

system. *International Journal on Advances in Intelligent Systems*, 2(1):181–191, 2009.

[47] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[48] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, 1992.

[49] P. Martin-Löf. Constructive mathematics and computer programming. In L.J. Cohen, J. Los, H. Pfeiffer, and K.-P. Podewski, editors, *LMPS 6*, pages 153–175. North-Holland, 1982.

[50] Mihhail Matskin, Riina Maigre, and Enn Tyugu. Compositional logical semantics for business process languages. In *ICIW*, page 38. IEEE Computer Society, 2007.

[51] Mihhail Matskin and Enn Tyugu. Strategies of structural synthesis of programs and its extensions. *Computers and Artificial Intelligence*, 20(1), 2001.

[52] James McDonald and John Anton. Specware - producing software correct by construction, 2001.

[53] Merik Meriste, Tõnis Kelder, and Jüri Helekivi. Towards multi-agent models of domain-specific languages. In Hele-Mai Haav and Ahto Kalja, editors, *BalticDB&IS*, pages 201–214. Institute of Cybernetics at Tallin Technical University, 2002.

[54] Merik Meriste, Tõnis Kelder, Jüri Helekivi, and Leo Motus. Domain-specific language agents. In Pekka Kilpeläinen and Niina Päivinen, editors, *SPLST*, pages 82–90. University of Kuopio, Department of Computer Science, 2003.

[55] Merik MERISTE and Jaan PENJAM. Attributed finite automata. Technical Report CS 23, Institute of Cybernetics, 1991.

[56] Merik Meriste and Jaan Penjam. Attributed models of executable specifications. In *PLILP*, pages 459–460, 1995.

[57] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[58] Grigori Mints. Resolution strategies for the intuitionistic logic. In *In Constraint Programming. NATO ASI Series F, v. 131*, pages 116–139. Springer, 1993.

[59] Grigori Mints, Jan M. Smith, and Enn Tyugu. Type-theoretical semantics of some declarative languages. In Barzdins and Bjørner [3], pages 18–32.

[60] Grigori Mints and Enn Tyugu. Justification of the structural synthesis of programs. *Science of Computer Programming*, 2(3):215–240, 1982.

[61] Grigori Mints and Enn Tyugu. Propositional logic programming and priz system. *J. Log. Program.*, 9(2&3):179–193, 1990.

[62] Grigori Mints and Enn Tyugu. The programming system PRIZ. In Barzdins and Bjørner [3], pages 1–17.

[63] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory: an introduction*. Clarendon Press, New York, NY, USA, 1990.

[64] Object Management Group: Model Driven Architecture. http://www.omg.org/mda.

[65] Jens Otten. ileantap: An intuitionistic theorem prover. In Didier Galmiche, editor, *TABLEAUX*, volume 1227 of *Lecture Notes in Computer Science*, pages 307–312. Springer, 1997.

[66] Jens Otten. Clausal connection-based theorem proving in intuitionistic first-order logic. In Bernhard Beckert, editor, *TABLEAUX*, volume 3702 of *Lecture Notes in Computer Science*, pages 245–261. Springer, 2005.

[67] Jukka Paakki. Attribute grammar paradigms – a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[68] Terence J Parr, T. J. Parr, and R W Quong. Antlr: A predicated-ll(k) parser generator, 1995.

[69] Jaan Penjam. Computational and attribute models of formal languages. *Theor. Comput. Sci.*, 71(2):241–264, 1990.

[70] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[71] Joseph Porter, Peter Volgyesi, Nicholas Kottenstette, Harmon Nine, Gabor Karsai, and Janos Sztipanovits. An experimental model-based rapid prototyping environment for high-confidence embedded software. In *20th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP'09)*, Paris, France, 06/2009 2009.

[72] Ando Saabas. A framework for design and implementation of visual languages. Master's thesis, Tallinn University of Technology, 2004.

[73] Jelena Sanko and Jaan Penjam. Differential evolutionary approach guided by the functional constraint network to solve program synthesis problem. In *Proceedings of IEEE World Congress on Computational Intelligence*, 2010.

[74] Bran Selic. A systematic approach to domain-specific language design using uml. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 2–9, Washington, DC, USA, 2007. IEEE Computer Society.

[75] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Software Eng.*, 16(9):1024–1043, 1990.

[76] Douglas R. Smith. KIDS - a knowledge-based software development system. In M. Lowry and R. McCartney, editors, *Automating Software Design*, pages 483–514. MIT Press, 1991.

[77] Yellamraju V. Srinivas, Yellamraju V. Srinivas, Richard Jüllig, and Richard Jullig. Specware: Formal support for composing software. In *In Mathematics of Program Construction*, pages 399–422. Springer-Verlag, 1995.

[78] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theor. Comput. Sci.*, 9:67–72, 1979.

[79] S. Doaitse Swierstra and Harald Vogt. Higher order attribute grammars. In Henk Alblas and Borivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 1991.

[80] Tanel Tammet. A resolution theorem prover for intuitonistic logic. In *CADE-13: Proceedings of the 13th International Conference on Automated Deduction*, pages 2–16, London, UK, 1996. Springer-Verlag.

[81] Juha-Pekka Tolvanen and Steven Kelly. Metaedit+: defining and using integrated domain-specific modeling languages. In Shail Arora and Gary T. Leavens, editors, *OOPSLA Companion*, pages 819–820. ACM, 2009.

[82] E. Tyugu. The structural synthesis of programs. In Andrei Ershov and Donald Knuth, editors, *Algorithms in Modern Mathematics and Computer Science*, volume 122 of *Lecture Notes in Computer Science*, pages 290–303. Springer Berlin / Heidelberg, 1981. 10.1007/3-540-11157-3_31.

[83] Enn Tyugu. *Knowledge-Based Programming*. Addison-Wesley, N.Y., 1988.

[84] Enn Tyugu. Higher order dataflow schemas. *Theor. Comput. Sci.*, 90(1):185–198, 1991.

[85] Enn Tyugu. *Algorithms and Architectures of Artificial Intelligence*. IOS Press, 2007.

[86] Enn Tyugu and Tarmo Uustalu. Higher-order functional constraint networks. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Proceedings NATO ASI on Constraint Programming, Pärnu, Estonia, 13–24 Aug 1993*, volume 131, pages 116–139. Springer-Verlag, Berlin, 1994.

[87] Enn Tyugu and Rando Valt. Visual programming in NUT. *Journal of Visual Languages and Computing*, 8(5-6):523–544, 1997.

[88] Tarmo Uustalu. Extensions of structural synthesis of programs. In U. H. Engberg, K. G. Larsen, and P. D. Mosses, editors, *Proceedings 6th Nordic Workshop*

on Programming Theory, NWPT'94, Aarhus, Denmark, 17–19 Oct 1994, pages 416–428. BRICS, Dept. of Computer Science, Univ. of Aarhus, Denmark, 1994.

[89] Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10:2002, 2001.

[90] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[91] Benjamin Volozh, Mihhail Matskin, Grigori Mints, and Enn Tyugu. The priz system and propositional calculus. *Cybernetics*, 18(6):777–788, 1982.

[92] Aare O. Vooglaid and Merik Meriste. Abstract attribute grammars. *Programming and Computer Software*, 8(5):242–251, 1982.

[93] Jos Warmer and Anneke Kleppe. *The object constraint language: precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[94] Jon Whittle. Transformations and software modeling languages: Automating transformations in uml. In Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors, *UML*, volume 2460 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2002.

List of Publications

- Gabriel Klein, Andres Ojamaa, Pavel Grigorenko, Marko Jahnke, and Enn Tyugu. Enhancing response selection in impact estimation approaches.*In: Concepts and Implementations for Innovative Military Communications and Information Technologies: (ed.) Amanowicz, Marek.* Warsaw: Military University of Technology, pp. 277–286, 2010.

- Pavel Grigorenko, Enn Tyugu. Higher-order attribute semantics of flat declarative languages. *Computing and Informatics*, 29(2), pp. 251–280, 2010.

- Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. Symbolic Query Exploration. *Formal Methods and Software Engineering : 11th International Conference on Formal Engineering Methods*, ICFEM 2009, Rio de Janeiro, Brazil, December 9–12, 2009, Proceedings. (eds.) Breitman, Karin; Cavalcanti, Ana. Berlin: Springer, (Lecture Notes in Computer Science; 5885), pp. 49–68, 2009.

- Riina Maigre, Pavel Grigorenko, Peep Küngas, Enn Tyugu. Stratified composition of web services. In: *Proceedings of the Eighth Joint Conference on Knowledge-Based Software Engineering*: (eds.) Virvou, Maria; Nakamura, Taichi. Amsterdam: IOS Press, (Frontiers in Artificial Intelligence and Applications; 180), pp. 49–58, 2008.

- Enn Tyugu, Pavel Grigorenko. Large-Scale Simulation Platform. *WSEAS Transactions on Computers*, No. 1, vol. 6, pp. 65–71, 2007.

- Pavel Grigorenko, Enn Tyugu. Deep Semantics of Visual Languages. In: *E. Tyugu, T. Yamaguchi (eds.) Knowledge-Based Software Engineering. Frontiers in Artificial Intelligence and Applications, vol. 140. IOS Press*, pp. 83–95, 2006.

- Pavel Grigorenko, Ando Saabas, Enn Tyugu. Visual Tool for Generative Programming. *In: Proceedings of the 10th European Software Engineering Conference: held jointly with 13th ACM SIGSOFT International Symposium on*

Foundations of Software Engineering ESEC/FSE-13. New York: ACM Press, pp. 249–252, 2005.

- Pavel Grigorenko, Ando Saabas, Enn Tyugu. COCOVILA – Compiler-Compiler for Visual Languages. *Proc. of the 5th Workshop on Language Descriptions, Tools and Applications, 2005* (Edinburgh, April 2005), v. 141, n. 4 of Electron. Notes in Theor. Comput. Sci., pp. 137–142. Elsevier, 2005.

Curriculum Vitae

Personal data

Name	Pavel Grigorenko
Date and place of birth	6.08.1982, Tallinn
Citizenship	Estonian

Contact information

Address	Akadeemia tee 21, 12618 Tallinn, Estonia
Phone	+372 620 4240
E-mail	pavelg@cs.ioc.ee

Education

2006 – 2010	Tallinn University of Technology, PhD studies in computer science
2004 – 2006	Tallinn University of Technology, MSc in computer science
2000 – 2004	Tallinn University of Technology, BSc in computer science

Language skills

Russian	–	native
English	–	fluent
Estonian	–	fluent

Professional Employment

2006 – ...	Institute of Cybernetics at TUT; Researcher
2009 – 2009	Microsoft Research, Redmond; Research intern
2005 – 2006	Institute of Cybernetics at TUT; Engineer
2001 – 2007	FutureTrade Estonia; Software Engineer

Defended theses

BSc thesis (2004):	"Program synthesis in Java environment", (sup) Enn Tyugu
MSc thesis (2006):	"Attribute semantics of visual languages", (sup) Enn Tyugu

Honours & Awards

Estonian Academy of Sciences student theses competition 2006 – I prize

Main areas of scientific work/Current research topics

Artificial intelligence, declarative specification languages, synthesis of programs, automated theorem proving

Elulookirjeldus

Isikuandmed

Ees- ja perekonnanimi	Pavel Grigorenko
Sünniaeg ja -koht	6.08.1982, Tallinn
Kodakondsus	Eesti

Kontaktandmed

Aadress	Akadeemia tee 21, 12618 Tallinn, Estonia
Telefon	+372 620 4240
E-post	pavelg@cs.ioc.ee

Hariduskäik

2006 – 2010	Tallinna Tehnikaülikool, doktoriõpe
2004 – 2006	Tallinna Tehnikaülikool, magistrikraad
2000 – 2004	Tallinna Tehnikaülikool, bakalaureusekraad

Keelteoskus

Vene keel	–	emakeel
Inglise keel	–	kõrgtase
Eesti keel	–	kõrgtase

Teenistuskäik

2006 – ...	TTÜ Küberneetika Instituut; Teadur
2009 – 2009	Microsoft Research, Redmond; Teadus intern
2005 – 2006	TTÜ Küberneetika Instituut; Insener
2001 – 2007	FutureTrade Estonia; Tarkvara Insiner

Kaitstud lõputööd

Bakalaureusetöö (2004):	“Programmide süntees Java keskkonnas”, (juh) Enn Tõugu
Magistritöö (2006):	“Visuaalsete keelte atribuutsemantika”, (juh) Enn Tõugu

Teaduspreemiad ja -tunnustused

Eesti Teaduste Akadeemia üliõpilastööde võistlus 2006 – I auhind

Teadustöö põhisuunad

Tehisintellekt, deklaratiivsed spetsifitseerimiskeeled, programmide süntees, automaatne teoreemide tõestamine

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational modelling of a communication office. 1992.

2. **Kalle Tammemäe**. Control intensive digital system synthesis. 1997.

3. **Eerik Lossmann**. Complex signal classification algorithms, based on the third-order statistical models. 1999.

4. **Kaido Kikkas**. Using the Internet in rehabilitation of people with mobility impairments – case studies and views from Estonia. 1999.

5. **Nazmun Nahar**. Global electronic commerce process: business-to-business. 1999.

6. **Jevgeni Riipulk**. Microwave radiometry for medical applications. 2000.

7. **Alar Kuusik**. Compact smart home systems: design and verification of cost effective hardware solutions. 2001.

8. **Jaan Raik**. Hierarchical test generation for digital circuits represented by decision diagrams. 2001.

9. **Andri Riid**. Transparent fuzzy systems: model and control. 2002.

10. **Marina Brik**. Investigation and development of test generation methods for control part of digital systems. 2002.

11. **Raul Land**. Synchronous approximation and processing of sampled data signals. 2002.

12. **Ants Ronk**. An extended block-adaptive Fourier analyser for analysis and reproduction of periodic components of band-limited discrete-time signals. 2002.

13. **Toivo Paavle**. System level modeling of the phase locked loops: behavioral analysis and parameterization. 2003.

14. **Irina Astrova**. On integration of object-oriented applications with relational databases. 2003.

15. **Kuldar Taveter**. A multi-perspective methodology for agent-oriented business modelling and simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected issues of modeling, verification and testing of digital systems. 2004.

18. **Ander Tenno**. Simulation and estimation of electro-chemical processes in maintenance-free batteries with fixed electrolyte. 2004.

19. **Oleg Korolkov**. Formation of diffusion welded Al contacts to semiconductor silicon. 2004.

20. **Risto Vaarandi**. Tools and techniques for event log analysis. 2005.

21. **Marko Koort**. Transmitter power control in wireless communication systems. 2005.

22. **Raul Savimaa**. Modelling emergent behaviour of organizations. Time-aware, UML and agent based approach. 2005.

23. **Raido Kurel**. Investigation of electrical characteristics of SiC based complementary JBS structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo.** Adaptive secure data transmission method for OSI level I. 2005.

26. **Deniss Kumlander**. Some practical algorithms to solve the maximum clique problem. 2005.

27. **Tarmo Veskioja**. Stable marriage problem and college admission. 2005.

28. **Elena Fomina**. Low power finite state machine synthesis. 2005.

29. **Eero Ivask**. Digital test in WEB-based environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian large vocabulary speech recognition. 2006.

32. **Erki Eessaar**. Relational and object-relational database management systems as platforms for managing softwareengineering artefacts. 2006.

33. **Rauno Gordon**. Modelling of cardiac dynamics and intracardiac bioimpedance. 2007.

34. **Madis Listak**. A task-oriented design of a biologically inspired underwater robot. 2007.

35. **Elmet Orasson**. Hybrid built-in self-test. Methods and tools for analysis and optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural networks based identification and control of nonlinear systems: ANARX model based approach. 2007.

37. **Toomas Kirt**. Concept formation in exploratory data analysis: case studies of linguistic and banking data. 2007.

38. **Juhan-Peep Ernits**. Two state space reduction techniques for explicit state model checking. 2007.

39. **Innar Liiv**. Pattern discovery using seriation and matrix reordering: A unified view, extensions and an application to inventory management. 2008.

40. **Andrei Pokatilov**. Development of national standard for voltage unit based on solid-state references. 2008.

41. **Karin Lindroos**. Mapping social structures by formal non-linear information processing methods: case studies of Estonian islands environments. 2008.

42. **Maksim Jenihhin**. Simulation-based hardware verification with high-level decision diagrams. 2008.

43. **Ando Saabas**. Logics for low-level code and proof-preserving program transformations. 2008.

44. **Ilja Tšahhirov**. Security protocols analysis in the computational model – dependency flow graphs-based approach. 2008.

45. **Toomas Ruuben**. Wideband digital beamforming in sonar systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model based method for adaptive decomposition of the thoracic bio-impedance variations into cardiac and respiratory components.

48. **Vineeth Govind**. DfT-based external test and diagnosis of mesh-like networks on chips. 2009.

49. **Andres Kull**. Model-based testing of reactive systems. 2009.

50. **Ants Torim**. Formal concepts in the theory of monotone systems. 2009.

51. **Erika Matsak**. Discovering logical constructs from Estonian children language. 2009.

52. **Paul Annus**. Multichannel bioimpedance spectroscopy: instrumentation methods and design principles. 2009.

53. **Maris Tõnso**. Computer algebra tools for modelling, analysis and synthesis for nonlinear control systems. 2010.

54. **Aivo Jürgenson**. Efficient semantics of parallel and serial models of attack trees. 2010.

55. **Erkki Joasoon**. The tactile feedback device for multi-touch user interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing situation – awareness cognition and reasoning of ad-hoc network agents. 2010.